robotron

DCP

SOFTWARE
Dokumentation

Anleitung für den Assemblerprogrammierer
Teil I – Heft 2

Anleitung
fuer
den Assemblerprogrammierer
Teil 1 - Heft 2

VEB Robotron Buchungsmaschinenwerk
Karl-Marx-Stadt, 1987

Die vorliegende 1. Auflage der Dokumentation "Anleitung fuer den Assemblerprogrammierer" unter DCP 3.2 entspricht dem Stand vom 30.6.87 und unterliegt nicht dem Aenderungsdienst.

Nachdruck, jegliche Vervielfaeltigung oder Auszuege daraus sind unzulaessig.

Die Dokumentation wurde durch ein Kollektiv des

VEB Robotron Buchungsmaschinenwerk Karl-Marx-Stadt

erarbeitet.

Bitte senden Sie uns Ihre Hinweise, Kritiken, Wuensche oder Forderungen zur Dokumentation zu.

VEB Robotron Buchungsmaschinenwerk
Karl-Marx-Stadt
PSF 129
Karl-Marx-Stadt
9010

Die "Anleitung fuer den Assemblerprogrammierer" besteht aus zwei Teilen

Teil 1 enthaelt:

I.	CPU - Befehlsbeschreibung	Heft 1
II.	Assembler (MASM)	Heft 2

Teil 2 enthaelt:

III.	Editoren (EDLIN, BE)	Heft 3
IV.	Bibliotheksverwalter (LIB)	Heft 3
V.	Binder (LINK)	Heft 4
VI.	Debugger (SYMDEB)	Heft 4
VII.	MAKE	Heft 4

I N H A L T S V E R Z E I C H N I S

		Seite
II.	MASM	8
1.	Einleitung	8
1. 1.	Leistungsfaehigkeit des Assemblers	8
1. 2.	Ueberblick ueber diese Beschreibung	8
1. 3.	Metasprache	8
2.	Elemente des Assemblers	9
2. 1.	Einleitung	9
2. 2.	Zeichensatz	9
2. 3.	Ganze Zahlen	9
2. 4.	Realzahlen	11
2. 5.	Verschluesselte Realzahlen	11
2. 6.	gepackte Dezimalzahlen	12
2. 7.	Zeichen- und Zeichenkettenkonstanten	12
2. 8.	Namen	12
2. 9.	Reservierte Namen	12
2.10.	Anweisungen	14
2.11.	Kommentare	15
2.12.	COMMENT	16
3.	Programmstruktur	16
3. 1.	Einleitung	16
3. 2.	Quelldatei	17
3. 3.	Pseudooperationen zur Definition der Anweisungsliste	19
3. 4.	SEGMENT und ENDS	20
3. 4. 1.	Align-Typ	21
3. 4. 2.	Combine-Typ	21
3. 4. 3.	Class-Typ	22
3. 4. 4.	Programmbeispiel	25
3. 4. 5.	Segmentverschachtelung	26
3. 5.	END	27
3. 6.	GROUP	28
3. 7.	ASSUME	30
3. 8.	ORG	30
3. 9.	EVEN	31
3.10.	PROC und ENDP	32
4.	Typen und Vereinbarungen	33
4. 1.	Einleitung	33
4. 2.	Markenvereinbarungen	33
4. 2. 1.	Definition einer NEAR-Marke	34
4. 2. 2.	Prozedurmarken	34
4. 3.	Datenvereinbarungen	35
4. 3. 1.	DB-Anweisung	35
4. 3. 2.	DW-Anweisung	36
4. 3. 3.	DD-Anweisung	38
4. 3. 4.	DQ-Anweisung	39
4. 3. 5.	DT-Anweisung	40

		Seite
4. 3. 6.	DUP-Operator	45
4. 4.	Symbolvereinbarungen	42
4. 4. 1.	Gleichheitszeichen (=)	43
4. 4. 2.	EQU	43
4. 4. 3.	LABEL	44
4. 5.	Typdefinitionen	45
4. 5. 1.	STRUC und ENDS	45
4. 5. 2.	RECORD	47
4. 6.	Struktur- und Recordvereinbarung	48
4. 6. 1.	Strukturvereinbarung	48
4. 6. 2.	Recordvereinbarung	50
5.	Operanden und Ausdruecke	52
5. 1.	Einleitung	52
5. 2.	Operanden	52
5. 2. 1.	Konstanten	53
5. 2. 2.	Direktadresse	53
5. 2. 3.	Verschiebliche Operanden	53
5. 2. 4.	Speicherplatzzaehleroperand	54
5. 2. 5.	Registeroperand	55
5. 2. 6.	Basisoperand	56
5. 2. 7.	Indexoperand	57
5. 2. 8.	Basis-Index-Operand	58
5. 2. 9.	Strukturoperand	59
5. 2.10.	Recordoperand	61
5. 2.11.	Record-Feld-Operand	61
5. 3.	Operatoren und Ausdruecke	62
5. 3. 1.	Arithmetische Operatoren	63
5. 3. 2.	SHR- und SHL-Operator	64
5. 3. 3.	Vergleichsoperatoren	65
5. 3. 4.	Bitoperatoren	66
5. 3. 5.	Indexoperator	66
5. 3. 6.	PTR-Operator	67
5. 3. 7.	Segmentpraefix	69
5. 3. 8.	Struktur-Feldname-Operator	69
5. 3. 9.	SHORT-Operator	70
5. 3.10.	THIS-Operator	70
5. 3.11.	HIGH- und LOW-Operator	71
5. 3.12.	SEG-Operator	71
5. 3.13.	OFFSET-Operator	72
5. 3.14.	TYPE-Operator	72
5. 3.15.	.TYPE-Operator	73
5. 3.16.	LENGTH-Operator	74
5. 3.17.	SIZE-Operator	74
5. 3.18.	WIDTH-Operator	74
5. 3.19.	MASK-Operator	75
5. 4.	Bestimmung von Ausdruecken und Rangfolge der Operatoren	75
5. 5.	Vorwaertsreferenzen	77
5. 6.	Strenge Typpruefung fuer Speicheroperanden	79
6.	Vereinbaren von Globalen	79
6. 1.	Einleitung	79
6. 2.	PUBLIC	80

		Seite
6. 3.	EXTRN	80
6. 4.	Programmbeispiel	81
7.	Bedingte Pseudooperationen	83
7. 1.	Einleitung	83
7. 2.	Pseudooperationen fuer die bedingte Uebersetzung	83
7. 2. 1.	IF und IFE	84
7. 2. 2.	IF1 und IF2	85
7. 2. 3.	IFDEF und IFNDEF	85
7. 2. 4.	IFB und IFNB	86
7. 2. 5.	IFIDN und IFDIF	87
7. 3.	Bedingte Fehlerpseudooperationen	88
7. 3. 1.	.ERR, .ERR1 und .ERR2	89
7. 3. 2.	.ERRE und .ERRNZ	90
7. 3. 3.	.ERRDEF und .ERRNDEF	90
7. 3. 4.	.ERRB und .ERRNB	91
7. 3. 5.	.ERRIDN und .ERRDIF	92
8.	Makroprogrammierung	92
8. 1.	Einleitung	92
8. 2.	Makro-Pseudooperationen	93
8. 2. 1.	MACRO und ENDM	94
8. 2. 2.	Makroaufrufe	96
8. 2. 3.	LOCAL	97
8. 2. 4.	PURGE	99
8. 2. 5.	REPT und ENDM	99
8. 2. 6.	IRP und ENDM	100
8. 2. 7.	IRPC und ENDM	101
8. 2. 8.	EXITM	102
8. 3.	Makro-Operatoren	103
8. 3. 1.	Ersetzungsoperator	103
8. 3. 2.	Literal-Text-Operator	105
8. 3. 3.	Literal-Zeichen-Operator	105
8. 3. 4.	Ausdruck-Operator	105
8. 3. 5.	Makro-Kommentar	106
9.	Pseudooperationen zur Dateisteuerung	107
9. 1.	Einleitung	107
9. 2.	INCLUDE	108
9. 3.	.RADIX	108
9. 4.	%OUT	110
9. 5.	NAME	110
9. 6.	TITLE	111
9. 7.	SUBTTL	111
9. 8.	PAGE	112
9. 9.	.LIST und .XLIST	113
9.10.	.SFCOND, .LFCOND und .TFCOND	113
9.11.	.LALL, .XALL und .SALL	114
9.12.	.CREF und .XCREF	116
10.	Abarbeiten des Makroassemblers	116
10. 1.	Einleitung	116
10. 2.	Starten und Verwenden von MASM	116

		Seite
10. 2. 1.	Aufrufen des Assemblers mit Hilfe von Eingabeaufforderungen	117
10. 2. 2.	Aufrufen des Assemblers mit Befehlszeile	119
10. 3.	Verwenden von MASM-Schaltern	121
10. 3. 1.	Segmente in alphabetischer Reihenfolge schreiben	122
10. 3. 2.	Segmente in der Quellkodereihenfolge schreiben	123
10. 3. 3.	Setzen Dateipuffergroesse	123
10. 3. 4.	Erzeugen einer Pass 1-Liste	123
10. 3. 5.	Definieren Assemblersymbol	124
10. 3. 6.	Festlegen eines Suchpfades fuer INCLUDE-Dateien	125
10. 3. 7.	Gross-/Kleinschreibung bei Namen	125
10. 3. 8.	Gross-/Kleinschreibung bei Eintrittspunkten und externen Namen	126
10. 3. 9.	Namen in Grossbuchstaben wandeln	126
10. 3.10.	Unterdruecken der Tabellen in der Assemblerliste	127
10. 3.11.	Pruefen auf unzulaessigen Kode	127
10. 3.12.	Kode fuer einen Gleitkommaprozessor erzeugen	127
10. 3.13.	Kode fuer einen Gleitkommaemulator erzeugen	128
10. 3.14.	Zusaetzliche Assemblerstatistik anzeigen	128
10. 3.15.	Falsche Bedingungen listen	129
10. 3.16.	Fehlerzeilen auf dem Bildschirm anzeigen	130
10. 3.17.	Crossreferenzdatei spezifizieren	130
10. 3.18.	Assemblerliste spezifizieren	130
10. 3.19.	Meldungen bei erfolgreichem Assemblieren unterdruecken	131
10. 4.	Hinweise zum lesen der Assemblerliste	131
10. 4. 1.	Hinweise zum Lesen des Kodes der Assembler-liste	131
10. 4. 2.	Hinweise zum Lesen einer Makrotabelle	141
10. 4. 3.	Hinweise zum Lesen einer Struktur- und Recordtabelle	141
10. 4. 4.	Hinweise zum Lesen einer Segment- und Gruppentabelle	142
10. 4. 5.	Hinweise zum Lesen der Symboltabelle	143
10. 4. 6.	Hinweise zum Lesen einer Pass 1-Liste	145

		Seite
11.	Crossreferenz	147
11. 1.	Einleitung	147
11. 2.	Verwenden von CREF	148
11. 2. 1.	Erzeugen einer Crossreferenzdatei	148
11. 2. 2.	Erzeugen einer Crossreferenzliste unter Verwendung von Eingabeaufforderungen	149
11. 2. 3.	Erzeugen einer Crossreferenzliste unter Verwendung der Befehlszeile	150
11. 3.	Das Format einer Crossreferenzliste	151
Anhang A	MASM-Fehlermeldungen	156
Anhang B	MASM-Pseudooperationen	170
Anhang C	MASM-Operatoren	177
Anhang D	CREF-Fehlermedlungen	181
Anhang E	Exit-Kodes fuer MASM und CREF	182

II. MASM

1. Einleitung

1.1. Leistungsfaehigkeit des Assemblers

In diesem Teil werden die Syntax und die Struktur der Assemblersprache fuer MASM beschrieben.

MASM hat einen umfangreichen Satz an Assembler-Pseudooperationen, mit deren Hilfe man die segmentierte Architektur des Mikroprozessors vollstaendig beherrschen kann.

Es wird eine grosse Anzahl von Datentypen der Operanden erlaubt, so zum Beispiel ganze Zahlen, Zeichenketten, gepackte Dezimalzahlen, Gleitkommazahlen, Strukturen und Records.

Durch den Assembler werden verschiebliche Objektmoduln erzeugt, die mittels Programmverbinder LINK zu ausfuehrbaren Programmen verbunden werden koennen. Die von MASM erzeugten Objektmoduln sind kompatibel zu vielen von Compilern hoeherer Programmiersprachen erzeugten Objektmoduln.

MASM bietet einen umfangreichen Satz Pseudooperationen fuer die Makroprogrammierung und fuer die bedingte Assemblierung.

Ein wesentliches Merkmal von MASM ist die strenge Syntaxpruefung aller Anweisungen, einschliesslich der Spezifikationen fuer Speicheroperanden. Dadurch werden fragwuerdige Operanden, die zu Fehlern oder unerwuenschten Ergebnissen fuehren koennten, erkannt.

1.2. Ueberblick ueber diese Beschreibung

Hiermit liegt eine detailierte Beschreibung der Syntax und der Struktur der Assemblersprache vor. Diese Beschreibung ist nicht geeignet, die Assemblerprogrammierung zu erlernen. Die Erlaeuterung des Mikroprozessors und die Erklaerung der Maschinenbefehle erfolgt in einem anderen Abschnitt.

Die Kapitel 2 bis 9 stellen die Sprachbeschreibung, das Kapitel 10 die Anwendungsbeschreibung des Assemblers dar. Im Kapitel 11 werden Erlaeuterungen zur Crossreferenz gegeben.

1.3. Metasprache

[...] Die Eingabe ist wahlfrei. Es ist zu beachten, dass diese Klammern fettgedruckt sind. Einfache Klammern haben andere Bedeutung, sie muessen geschrieben werden.

<...> Ein kleingeschriebener Text in spitzen Klammern muss vom Anwender durch eine konkrete Zeichenkette ersetzt werden. Normalerweise sind dabei die spitzen

Klammern nicht mit zu schreiben. Wenn sie geschrieben werden muessen, ist dies ausdruecklich vermerkt.

<GROSSBUCHSTABEN>
Es ist die genannte Taste zu druecken.

{...|...} Auswahl zwischen 2 oder mehreren Eintragungen. Eine der Eintragungen muss verwendet werden.

[,...] Die Eintragung kann wiederholt werden.

Alle anderen Teile der Syntaxbeschreibung sind unveraendert zu schreiben.

2. Elemente des Assemblers

2.1. Einleitung

Alle Assemblerprogramme bestehen aus einer oder mehreren Anweisungen und Kommentaren. Eine Anweisung oder ein Kommentar ist eine Kombination von Zeichen, Zahlen und Namen.

Namen und Zahlen werden zum Identifizieren von Werten in Anweisungen verwendet. Aus Zeichen werden Namen, Zahlen oder Zeichenkonstanten gebildet.

2.2. Zeichensatz

Fuer MASM ist der folgende Zeichensatz gueltig:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
? @ _ $ : . [ ] ( ) < > { }
+ - / * & % ! ' ` | \ = # ^ ; , " 
```

2.3. Ganze Zahlen

Syntax:

```
<ziffern>
<ziffern>B
<ziffern>Q
<ziffern>O
<ziffern>D
<ziffern>H
<ziffern>R
```

Eine ganze Zahl ist eine Kombination von Binaer-, Oktal-, Dezimal- oder Hexadezimalziffern und einer wahlfreien Zahlenbasis. <ziffern> sind eine oder mehrere Ziffern einer speziellen Zahlenbasis: B, Q, O, D oder H. Der Bezeichner fuer Realzahlen kann ebenfalls verwendet werden. Wurde keine Zahlenbasis angegeben,

nimmt der Assembler die aktuelle Standardzahlenbasis an (dezimal, wenn diese nicht mit der Pseudooperation .RADIX geaendert wurde). Die Zahlenbasis kann in grossen oder kleinen Buchstaben geschrieben werden. In den Beispielen in dieser Sprachbeschreibung wurden kleine Buchstaben verwendet.

In der folgenden Tabelle werden die moeglichen Ziffern fuer jede Zahlenbasis angegeben:

Zahlenbasis	Zahlentyp	Ziffern
B	Binaer	0 1
Q oder O	Oktal	0 1 2 3 4 5 6 7
D	Dezimal	0 1 2 3 4 5 6 7 8 9
H	Hexadezimal	0 1 2 3 4 5 6 7 8 9 A B C D E F
R	Realzahl	0 1 2 3 4 5 6 7 8 9 A B C D E F

Hexadezimalziffern muessen immer mit einer Dezimalziffer (0 bis 9) beginnen. Wenn erforderlich, muss links eine fuehrende Null hinzugefuegt werden, um eine Hexadezimalzahl, die mit einem Buchstaben beginnt von einem Symbol unterscheiden zu koennen. So wird zum Beispiel 0ABCh als Hexadezimalzahl interpretiert, aber ABCh als Symbol. Die Hexadezimalziffern A bis F koennen als grosse oder kleine Buchstaben geschrieben werden.

Der Bezeichner R fuer Realzahlen kann nur in Verbindung mit Hexadezimalziffern verwendet werden, die aus 8, 16 oder 20 signifikanten Ziffern bestehen (eine fuehrende Null kann hinzugefuegt werden).

Die maximale Anzahl von Ziffern fuer eine ganze Zahl haengt von der Anweisung oder Pseudooperation ab, in der diese ganze Zahl geschrieben wird.

Die Standardzahlenbasis kann durch die Pseudooperation .RADIX spezifiziert werden (siehe auch Abschnitt 9.3.).

Beispiele:

```
01011010b     132q     5Ah     90d
01111b        170      0Eh     15d
```

2.4. Realzahlen

Syntax:

[(+|-)]<ziffern>.<ziffern>[E[(+|-)]<ziffern>]

Eine Realzahl ist eine Zahl, die aus folgenden Teilen besteht:

- einer ganzen Zahl
- einem gebrochenen Teil
- und einem Exponenten.

<ziffern> kann eine Kombination von Dezimalziffern sein. Die <ziffern> vor dem Dezimalpunkt stellen die ganze Zahl dar. Die dem Punkt folgenden <ziffern> sind der gebrochene Teil. Die <ziffern> nach dem Kennzeichen fuer den Exponenten E sind der Exponent. Dieser ist wahlfrei. Ein gegebener Exponent kann ein Vorzeichen (+|-) haben.

Realzahlen koennen nur mit den Pseudooperationen DD, DQ oder DT verwendet werden. Die maximale Ziffernanzahl der Zahl und die maximale Groesse des Exponenten haengen von der Pseudooperation ab (siehe dazu auch die Abschnitte 4.3.3., 4.3.4. und 4.3.5. in dieser Beschreibung).

Beispiele:

```
65.78
6.578E1
6578.0E-2
```

2.5. Verschluesselte Realzahlen

Syntax:

<ziffern>R

Eine verschluesselte Realzahl besteht aus 8, 16 oder 20 Hexadezimalziffern, die eine Realzahl in verschluesseltem Format darstellen.

Eine verschluesselte Realzahl hat ein Vorzeichen, einen vorangestellten Exponenten und eine Mantisse. Diese Werte sind in Bitfeldern innerhalb der Zahl verschluesselt. Die genaue Groesse und die Bedeutung jedes Bitfeldes haengt von der Bitanzahl in der Zahl ab. Die Ziffern muessen Hexadezimalziffern sein. Die Zahl muss mit einer Dezimalziffer (0 bis 9) beginnen und am Ende dieser Realzahl muss der Bezeichner R stehen.

Eine verschluesselte Realzahl kann nur mit den Pseudooperationen DD, DQ oder DT verwendet werden. Diese Zahl muss 8, 16 oder 20 Ziffern haben. Wenn eine Null vorangestellt werden muss, betraegt die Ziffernanzahl 9, 17 oder 21. (Siehe dazu auch die Abschnitte 4.3.3., 4.3.4. und 4.3.5.)

Beispiele:

```
DD      3F800000r               ;1.0 fuer DD
DQ      3FF0000000000000r       ;1.0 fuer DQ
```

2.6. Gepackte Dezimalzahlen

Syntax:

```
[{+|-}]<ziffern>
```

Eine gepackte Dezimalzahl ist eine ganze Dezimalzahl, die in gepacktem Format gespeichert wird. Gepackte Dezimalzahlen haben ein Vorzeichenbyte und 9 Wertebytes. Jedes Wertebyte enthaelt 2 Dezimalziffern. Das hoechstwertige Bit des Vorzeichenbytes ist 0 fuer positive Werte und 1 fuer negative Werte.

Gepackte Dezimalzahlen haben dasselbe Format wie andere ganze Dezimalzahlen (Ganze Zahlen) mit der Ausnahme, dass sie zusaetzlich ein wahlfreies Vorzeichen haben und nur mit der Pseudooperation DT definiert werden koennen.

Eine gepackte Dezimalzahl kann nicht mehr als 18 Ziffern haben.

Beispiele:

```
DT  1234567890      ;intern: 000000000001234567890h
DT  -1234567890     ;intern: 800000000001234567890h
```

2.7. Zeichen- und Zeichenkettenkonstanten

Syntax:

```
'<zeichen>'
"<zeichen>"
```

Eine Zeichenkonstante enthaelt ein einzelnes ASCII-Zeichen, eine Zeichenkettenkonstante zwei oder mehr ASCII-Zeichen. Die ASCII-Zeichen muessen in einfache Hochkommas oder in Anfuehrungszeichen eingeschlossen werden. Bei Zeichen- oder Zeichenkettenkonstanten bleibt die gewaehlte Schreibweise (kleine oder grosse Buchstaben) immer erhalten!

Soll das begrenzende Zeichen Bestandteil der Zeichenkettenkonstante sein, muss es innerhalb der Konstante doppelt geschrieben werden. Links und rechts muss das gleiche Begrenzungszeichen stehen.

Beispiele:

```
'a'                     ;a
'ab'                    ;ab
"a"                     ;a
"Das ist eine Meldung"  ;Das ist eine Meldung
'File can''t found'     ;File can't found
"File can't found"      ;File can't found
```

2.8. Namen

Syntax:

```
<zeichen>
```

Ein Name ist eine Folge von Buchstaben, Ziffern und speziellen Zeichen (_, %, ?, $, @) und wird in Assembleranweisungen als Marke, Variable oder Symbol verwendet.

Namen werden nach folgenden Regeln gebildet:

- Das erste Zeichen eines Namens muss ein Buchstabe, ein Unterstreichstrich (_), ein Fragezeichen (?), ein Punkt (.) oder das kommerzielle a (@) sein.

- Ein Name kann mit einem Punkt beginnen, aber es darf kein Punkt innerhalb des Namens auftreten.

- Ein Name kann grosse und kleine Buchstaben gemischt enthalten. Alle kleinen Buchstaben werden durch den Assembler in grosse gewandelt. Die Kleinschreibung bleibt erhalten, wenn der /ML-Schalter verwendet wurde oder wenn der Name mit PUBLIC oder EXTRN definiert und der /MX-Schalter waehrend der Uebersetzung gesetzt war.

- Die ersten 31 Zeichen eines Namens sind signifikant. Alle weiteren Zeichen werden ignoriert.

- Ein Name darf nicht mit einem reservierten Namen uebereinstimmen. Verboten sind zum Beispiel: AX, ?, $, mov, PAGE usw.

Beispiele:

```
up3
Feld
_main
.sym1
```

2.9. Reservierte Namen

Ein reservierter Name ist jeder Name mit einer speziellen vorgegebenen Bedeutung fuer den Assembler.

Reservierte Namen schliessen Anweisungs- und Pseudooperations-mnemoniken, Registernamen und Operatornamen ein. Diese Namen koennen nur so, wie sie definiert sind, verwendet werden. Sie duerfen nicht anders benutzt werden.

Alle Kombinationen von grossen und kleinen Buchstaben werden als der gleiche Name behandelt. Zum Beispiel sind LENGTH und Length die gleichen Namen fuer den Operator LENGTH.
In der folgenden Tabelle werden alle reservierten Namen mit Ausnahme der Befehlsmnemoniken aufgelistet:

Reservierte Namen

.186	DI	.ERRNZ	LENGTH	.SALL
.286c	DL	ES	.LFCOND	SEG
.286p	DQ	EVEN	.LIST	SEGMENT
.287	DS	EXITM	LOCAL	.SFCOND
.8086	DT	EXTRN	LOW	SHL
.8087	DW	FAR	LT	SHORT
=	DWORD	GE	MACRO	SHR
AH	DX	GROUP	MASK	SI
AL	ELSE	GT	MOD	SIZE
AND	END	HIGH	NAME	SP
ASSUME	ENDIF	IF	NE	SS
AX	ENDM	IF1	NEAR	STRUC
BH	ENDP	IF2	NOT	SUBTTL
BL	ENDS	IFB	OFFSET	TBYTE
BP	EQ	IFDEF	OR	.TFCOND
BX	EQU	IFDIF	ORG	THIS
BYTE	.ERR	IFE	%OUT	TITLE
	.ERR1	IFIDN	PAGE	TYPE
_	.ERR2	IFNB	PROC	.TYPE
COMMENT	.ERRB	IFNDEF	PTR	WIDTH
.CREF	.ERRDEF	INCLUDE	PUBLIC	WORD
CS	.ERRDIF	IRP	PURGE	.XALL
CX	.ERRE	IRPC	QWORD	.XCREF
DB	.ERRIDN	LABEL	.RADIX	.XLIST
DD	.ERRNB	.LALL	RECORD	XOR
DH	.ERRNDEF	LE	REPT	

2.10. Anweisungen

Syntax:

[<name>] <mnemonik> [<operanden>] [;<kommentar>]

Eine Anweisung besteht aus einem wahlfreien Namen, der Mnemonik eines Maschinenbefehls oder einer Pseudooperation, einem oder mehreren wahlfreien Operanden und wahlfreiem Kommentar. Eine Anweisung stellt eine Aktion des Assemblers dar, einen Maschinenbefehl, oder generiert ein oder mehrere Datenbytes.

Anweisungen werden nach folgenden Regeln gebildet:

- Eine Anweisung kann in jeder Spalte beginnen.
- Eine Anweisung darf nicht mehr als 128 Zeichen lang sein und kein eingebettetes 0Dh, 0Ah haben. Anders gesagt, das Fortsetzen einer Anweisung auf weiteren Zeilen ist nicht erlaubt.
- Alle Anweisungen mit Ausnahme der letzten in der Datei muessen durch 0Dh, 0Ah abgeschlossen sein.

Beispiele:

```
zaehler DB      0
        mov     ax,bx
        ASSUME  cs:_text,ds:DGROUP
druck   PROC    NEAR
```

2.11. Kommentare

Syntax:

```
;<text>
```

Ein Kommentar ist jede Kombination von Zeichen, eingeleitet durch ein Semikolon (;) und abgeschlossen durch ein 0Dh, 0Ah. Kommentare beschreiben die Aktion eines Programms an einem bestimmten Punkt. Sie werden vom Assembler ignoriert und haben fuer die Uebersetzung keine Bedeutung. Kommentare koennen ueberall im Programm stehen, auch auf der Zeile einer Anweisung.

Wenn der Kommentar auf einer Anweisungszeile steht, dann muss er rechts von Namen, Mnemonik und Operanden angeordnet werden. Ein nach einem Semikolon stehender Kommentar darf nicht ueber das Ende der Zeile, in der er beginnt, fortgesetzt werden. Das heisst, er darf kein eingebettetes 0Dh, 0Ah enthalten. Fuer sehr lange Kommentare kann die Pseudooperation COMMENT verwendet werden.

Beispiele:

```
;Dieser Kommentar steht allein auf einer Zeile
mov     ax,bx     ;Dieser Kommentar folgt einer Anweisung
;Kommentare koennen reservierte Woerter enthalten
```

2.12. COMMENT

Syntax:

```
COMMENT <begrenzer>
<text>
<begrenzer> [<text>]
```

Die Pseudooperation veranlasst den Assembler, allen <text> zwischen <begrenzer> und <begrenzer> als Kommentar zu behandeln. Das Zeichen <begrenzer> muss das erste vom Leerzeichen verschiedene Zeichen nach dem Schluesselwort COMMENT sein. <text> sind alle nachfolgenden Zeichen bis zum naechsten Auftreten von <begrenzer>. Der Text darf den Begrenzer nicht enthalten.

Die Pseudooperation COMMENT wird fuer mehrzeilige Kommentare verwendet. Text, der irgendwo auf derselben Zeile wie der zweite Begrenzer auftritt, wird vom Assembler ignoriert.

Beispiele:

```
comment *
Dieser Kommentar wird fortgesetzt
bis zum naechsten Stern
*
```

Das vorausgehende und das nachfolgende Beispiel zeigen, wie Textbloecke als Kommentar gekennzeichnet werden koennen.

```
comment +
Der Assembler ignoriert die Anweisung,
die dem letzten Begrenzer folgt
+ mov  ax,1
```

3. Programmstruktur

3.1. Einleitung

Durch die Pseudooperationen zur Programmstrukturierung laesst sich die Organisation definieren, die Programmkode und Daten beim Laden in den Speicher besitzen sollen.

Zu dieser Gruppe gehoeren folgende Pseudooperationen:

Pseudo-operation	Bedeutung
SEGMENT	Segmentdefinition
ENDS	Segmentende
END	Quelldateiende
GROUP	Segmente zu einer Gruppe zusammenfassen
ASSUME	Segmentregister zuweisen
ORG	Segmentanfang
EVEN	Segment ausrichten auf Speichergrenzen
PROC	Prozedurdefinition
ENDP	Prozedurende

3.2. Quelldatei

Jedes Assemblerprogramm wird aus einer oder mehreren Quelldateien erzeugt (Textdateien, die Anweisungen zur Definition von Programmdaten und Befehlen enthalten). MASM liest diese Quelldateien, uebersetzt die Anweisungen und erzeugt Objektmodule. LINK bereitet dann die Objektmodule fuer die Ausfuehrung vor.

Die Quelldateien muessen vom Standard-ASCII-Format sein: sie duerfen keine Steuerkodes enthalten und jede Zeile muss mit 0Dh, 0Ah abgeschlossen sein.

Die Anweisungen koennen in kleinen und grossen Buchstaben eingegeben werden. In dieser Sprachbeschreibung werden reservierte Woerter und Klassentypen in grossen Buchstaben geschrieben. Das ist aber keine generelle Forderung.

Alle Quelldateien haben die gleiche Form: Null oder mehr Programmsegmente gefolgt von einer END-Anweisung. Eine nur aus Makros, Strukturen und Records bestehende Quelldatei hat Null Segmente.

Die Pseudooperation END ist in jeder Quelldatei erforderlich. Sie zeigt das Ende der Quelldatei an. Sie dient ausserdem dazu, den Eintrittspunkt oder die Startadresse des Programms zu definieren.

Das folgende Beispiel erlaeutert das Format der Quelldatei. Es ist ein vollstaendiges Assemblerprogramm, das DCP-Funktionen

(oder Systemrufe) verwendet, um die Meldung **Hier meldet sich Ihr Computer** auf dem Bildschirm anzuzeigen.

Beispiel:

```
daten   SEGMENT                         ;Datensegment
meldung DB      "Hier meldet sich Ihr Computer",13,10,"$"
daten   ENDS

code    SEGMENT                         ;Kodesegment
        ASSUME  cs:code,ds:daten
beginn:                                 ;Eintrittspunkt
        mov     ax,daten                ;Datensegmentadresse
        mov     ds,ax                   ; ins DS-Register
        mov     dx,OFFSET meldung       ;Laden Adresse der
                                        ; Zeichenkette
        mov     ah,09h                  ;Call Anzeigen
        int     21h                     ; Zeichenkette
        mov     ah,4Ch                  ;Call Funktion
        int     21h                     ; Programmende
code    ENDS

stack   SEGMENT stack                   ;Stacksegment
        DW      64 DUP(?)               ;Definieren Platz
                                        ; fuer Stack
stack   ENDS

        END     beginn
```

Die folgenden Hauptmerkmale dieser Quelldatei sollten beachtet werden:

1. **Die Pseudooperationen SEGMENT und ENDS**, durch die die Segmente mit den Namen daten, code und stack definiert werden.

2. **Die Variable meldung im Datensegment**, die die anzuzeigende Zeichenkette definiert. Sie beinhaltet sowohl den Kode fuer CR/LF als auch das Waehrungszeichen ($), das von der DCP-Funktion "Anzeige Zeichenkette" benoetigt wird.

3. **Die Befehlsmarke beginn im Kodesegment**, die den Beginn der Programmbefehle kennzeichnet.

4. **Die DW-Anweisung im Stacksegment**, die den nichtinitialisierten Speicherplatz fuer den Programmstack definiert.

5. **Die ASSUME-Anweisung fuer das Daten- und das Kodesegment**, durch die spezifiziert wird, welche Segmentregister mit den innerhalb des Segments definierten Marken, Variablen und Symbolen verbunden werden. Eine ASSUME-Anweisung fuer das Stacksegment ist nicht notwendig, weil der Combine-Typ dem Assembler mitteilt, dass das Segment mit dem Register SS verbunden wird.

6. **Die ersten beiden Befehle, die die Adresse des Datensegments in das Register DS laden.** Diese Befehle sind fuer das Kode- und Stacksegment nicht erforderlich, weil die Kodesegmentadresse immer in das Register CS geladen wird und die Stacksegmentadresse automatisch in das Register SS, wenn der Combine-Typ stack verwendet wird.

7. **Die letzten beiden Befehle im Kodesegment, die die DOS-Funktion 4Ch aufrufen, um zu DOS zurueckzukehren.** Obwohl es andere Rueckkehrmoeglichkeiten gibt, ist diese die zu empfehlende fuer die meisten Assemblerprogramme.

8. **Die END-Anweisung,** die das Quelldateiende anzeigt und beginn als Eintrittspunkt des Programms festlegt.

3.3. Pseudooperationen zur Defintion der Anweisungsliste

Syntax:

```
.8086
.8087
.186
.286c
.286p
.287
```

Die Pseudooperationen zur Definition der Anweisungsliste fuer die Maschinenbefehle ermoeglichen Befehle fuer den gegebenen Mikroprozessor zu schreiben. Wenn eine solche Pseudooperation angegeben ist, wird vom Assembler jeder folgende zu diesem Prozessortyp gehoerende Befehl akzeptiert und uebersetzt.

Diese Pseudooperationen muessen am Anfang des Programms stehen. Sie sichern, dass alle Befehle einer Datei unter Verwendung der gleichen Anweisungsliste uebersetzt werden.

Die Anweisung .8086 ermoeglicht die Uebersetzung von Befehlen fuer die Prozessoren K1810 WM86, 8086 und 8088. Damit koennen Befehle fuer die Prozessoren 80186 und 80286 nicht uebersetzt werden.

Aehnlich verhaelt es sich bei .8087. Es koennen Befehle fuer den Gleitkomma-Koprozessor 8087, aber keine fuer den 80287 geltenden Befehle uebersetzt werden.

MASM uebersetzt standardmaessig die Befehle des K1810 WM86, des 8086 und 8087. Deshalb sind die Pseudooperationen .8086 und .8087 nicht erforderlich, wenn die Quelldatei nur solche Befehle enthaelt. Wird die Standardbefehlsliste nicht verwendet, sind die Programme unter Umstaenden nicht auf allen Prozessoren der Familie lauffaehig.

.186 erlaubt die Befehle des K1810 WM86, des 8086 und die zusaetzlichen Befehle des 80186.

Mit .286c sind die Befehle des K1810 WM86, des 8086 und die Befehle des **nichtgeschuetzten** Modus fuer den 80286 moeglich. Mit .286p koennen zusaetzlich zu den vorher genannten die Befehle des geschuetzten Modus des 80286 verwendet werden.

.287 laesst die Befehlsliste des Gleitkomma-Koprozessors 80287 zu. Solche Programme koennen nur auf dem 80286 abgearbeitet werden.

Sind die Pseudooperationen .8087 oder .287 in der Quelldatei enthalten, verlangt MASM die Angabe des /R- oder /E-Schalters in der Befehlszeile, um die Art der Uebersetzung der Gleitkommabefehle festzulegen.

Mit dem /R-Schalter wird Maschinenkode fuer die Gleitkommabefehle erzeugt. Der /E-Schalter bewirkt die Kodeausgabe fuer eine Gleitkomma-Emulator-Routine. Siehe dazu auch die Abschnitte 10.3.12. und 10.3.13.

3.4. SEGMENT und ENDS

Syntax:

```
<name> SEGMENT [<align>] [<combine>] ['<class>']
<name> ENDS
```

Durch die Pseudooperationen SEGMENT und ENDS wird Beginn und Ende eines Programmsegments markiert. Ein Programmsegment ist eine Menge von Befehlen und / oder Daten, deren Adressen alle relativ zu demselben Segmentregister sind.

<name> bezeichnet den Namen des Segments. Entweder ist der Name einmalig oder der Name eines bereits vorhandenen Segments. Segmente mit dem gleichen Namen werden als dasselbe Segment gewertet.

Die wahlfreien Parameter <align>-, <combine>- und '<class>'-Typ geben dem Programmverbinder Anweisungen, wie die Segmente auszugeben sind. Sie muessen in dieser Reihenfolge, aber nicht alle drei angegeben werden.

Beachte:

Man darf den byte- und word-align-Typ nicht mit den reservierten Woertern BYTE und WORD verwechseln. Letztere werden verwendet, um den Datentyp mit den Operatoren THIS und PTR zu spezifizieren.

Ebenso sollte der page-align-Typ und public-combine-Typ nicht mit den Pseudooperationen PAGE und PUBLIC verwechselt werden. Der Unterschied muesste aus dem Zusammenhang klar sein, da der align- und combine-Typ nur auf derselben Zeile, wie die Pseudooperation SEGMENT verwendet wird.

3.4.1. Align-Typ

Der wahlfreie Parameter align-Typ gibt die Ausrichtung des gegebenen Segments an. Diese Ausrichtung definiert einen Bereich von Speicheradressen, aus dem eine Startadresse fuer das Segment ausgewaehlt werden kann.

Der align-Typ kann einer der folgenden sein:

Align-Typ	Bedeutung
byte	jede Byteadresse kann verwendet werden
word	jede Wortadresse kann verwendet werden 2 Bytes / Wort (durch 2 teilbare Adresse)
para	jede Paragraphadresse kann verwendet werden 16 Bytes / Paragraph (durch 16 teilbare Adresse)
page	Seitenadressen koennen verwendet werden 256 Bytes / Seite (durch 256 teilbare Adresse)

Para ist der Standard-align-Typ. Die echte Startadresse wird erst beim Laden des Programms berechnet. Der Programmverbinder garantiert, dass die Ladeadresse an der entsprechenden Speichergrenze liegt.

3.4.2. Combine-Typ

Der wahlfreie Parameter combine-Typ definiert, wie Segmente gleichen Namens verbunden werden.

Der combine-Typ kann einer der folgenden sein:

public Verketten aller Segmente mit dem gleichen Namen zu einem einzigen zusammenhaengenden Segment. Alle Anweisungs- und Datenadressen in dem neuen Segment sind relativ zu einem einzigen Segmentregister und alle Offsets werden so angepasst, dass sie den Abstand zum Beginn des neuen Segments darstellen.

stack Verketten aller Segmente mit dem gleichen Namen zu einem einzigen zusammenhaengenden Segment. Dieser combine-Typ ist der gleiche wie public, mit der Ausnahme, dass alle Adressen in dem neuen Segment relativ zum Segmentregister SS sind. Das Stackpointerregister SP wird auf die Endeadresse des Segments initialisiert.

Fuer das Stacksegment sollte normalerweise der stack-Typ verwendet werden, da dann automatisch das SS-Register initialisiert wird. Ist dieser Typ fuer ein Stacksegment nicht angegeben worden, muss das Laden der Segmentadresse programmiert werden.

common Erzeugt sich ueberlagernde Segmente, wobei der Anfang aller Segmente mit demselben Namen dieselbe Adresse ist. Die Laenge des daraus resultierenden Bereichs ist gleich der Laenge des laengsten Segments. Alle Adressen in den Segmenten sind relativ zu derselben Basisadresse. Wurden Daten in mehr als einem Segment mit dem gleichen Namen vereinbart, so ersetzen die zuletzt vereinbarten Daten die vorher definierten.

memory Dieser Typ wird durch den Programmverbinder LINK exakt wie public behandelt. MASM erlaubt Segmente mit dem memory-Typ, obwohl LINK diesen Typ nicht unterstuetzt. Diese Eigenschaft wurde wegen der Kompatibilitaet mit anderen Programmverbindern vorgesehen.

at<adresse> Bewirkt, dass alle in diesem Segment definierten Marken und variablen Adressen relativ zu der gegebenen Adresse errechnet werden. <adresse> kann jeder gueltige Ausdruck sein. Er darf jedoch keine Vorwaertsreferenz enthalten, d. h. keinen Verweis auf ein Symbol, das spaeter in der Quelldatei definiert wird. Typisch fuer ein solches Segment ist, dass es weder Kode noch Daten enthaelt. Statt dessen stellt es eine Adressentabelle dar, die ueber Kode oder Daten ueberall im Speicher gelegt werden kann, beispielsweise ueber den Bildwiederholspeicher. Die Marken und Variablen koennen dann fuer symbolischen Zugriff auf feste Anweisungen und Daten verwendet werden.

Wenn kein combine-Typ angegeben ist, wird das Segment nicht mit anderen verbunden. Es wird angenommen, das es ein eigenes Segment ist.

Beachte!

Normalerweise sollte man als letztes im Programm ein Stacksegment vorsehen. Wurde kein Stacksegment angelegt, gibt der Programmverbinder eine Warnung aus. Diese Warnung kann ignoriert werden, wenn aus einem bestimmten Grund kein Stacksegment gewuenscht wird.

3.4.3. Class-Typ

Der wahlfreie Parameter class-Typ legt fest, welche Segmente in benachbarten Speicher geladen werden. Segmente gleichen Klassen-

namens werden hintereinander in den Speicher geladen. Alle Segmente einer Klasse werden vor den Segmenten einer anderen Klasse angeordnet. Der Klassenname muss in einfache Hochkommas (') eingeschlosssen werden. Gross- oder Kleinschreibung wird nicht beruecksichtigt, es sei denn, waehrend der Uebersetzung wurden die Schalter /MX oder /ML bzw. /NOIGNORECASE beim Programmverbinder angegeben.

Beachte:

Namen, die als Klassentypen fuer Segmente zugewiesen wurden, duerfen nicht fuer andere Symboldefinitionen in der Quelldatei verwendet werden. Es wird sonst ein Fehler generiert:

Symbol already different kind

(das Symbol ist schon in anderer Art vorhanden).

Wenn keine Klassentypen spezifiziert wurden, schreibt der Programmverbinder die Segmente in der gleichen Reihenfolge in die ausfuehrbare Datei, wie sie in den Objektdateien auftreten. Diese Reihenfolge wird immer eingehalten, wenn der Programmverbinder nicht zwei oder mehr Segmente mit gleichem Klassennamen vorfindet.

Segmente mit identischen Klassennamen gehoeren zu derselben Klasse und werden als benachbarte Bloecke in die ausfuehrbare Datei geschrieben.

Beispiel:

```
DATAX   SEGMENT  'DATA'
DATAX   ENDS

TEXT    SEGMENT  'CODE'
TEXT    ENDS

DATAZ   SEGMENT  'DATA'
DATAZ   ENDS
```

In diesem Programmfragment haben die beiden Segmente DATAX und DATAZ den gleichen Klassennamen 'DATA'. Das Ergebnis ist, dass beide Segmente hintereinander vor das Segment TEXT in die ausfuehrbare Datei geschrieben werden.

Alle Segmente gehoeren einer Klasse an. Segmente, fuer die nicht explizit ein Klassenname vergeben wurde, erhalten den Null-Klassennamen. Sie werden als benachbarte Bloecke mit den anderen Segmenten, die den Null-Klassennamen haben, geladen.

LINK enthaelt keine Begrenzung fuer die Anzahl oder die Groesse der Segmente einer Klasse. Die totale Groesse aller Segmente in einer Klasse kann 64 k Byte ueberschreiten.

LINK arbeitet die Moduln in der Reihenfolge ab, wie er sie in der Befehlszeile erhaelt. Man ist nicht immer in der Lage, Einfluss auf die Segmentladereihenfolge zu nehmen. Besteht beispielsweise ein Programm aus vier Segmenten, die in folgender Reihenfolge geladen werden sollen: CODE, DATA, CONST, STACK. Die Segmente CODE, CONST und STACK sind im ersten Modul des Programms, das DATA-Segment ist im zweiten Modul definiert. Link ordnet dann die Segmente nicht in der gewuenschten Reihenfolge, weil zuerst die im ersten Modul angelegten Segmente geladen werden.

Dieses Problem laesst sich durch Erzeugen und Uebersetzen eines Scheinprogramms vermeiden, welches nur leere Segmentdefinitionen in der gewuenschten Reihenfolge enthaelt. Diese uebersetzte Datei ist dann beim Verbinden als erste Objektdatei anzugeben.

Das folgende Beispiel zeigt ein solches Scheinprogramm mit festgelegter Ladereihenfolge der Segmente, die mit CODE, DATA, CONST und STACK benannt sind.

Beispiel:

```
CODE    SEGMENT para public 'CODE'
CODE    ENDS
DATA    SEGMENT para public 'DATA'
DATA    ENDS
CONST   SEGMENT para public 'CONST'
CONST   ENDS
STACK   SEGMENT para stack 'STACK'
STACK   ENDS
```

Dieses Programm muss die Definitionen fuer alle im Programm verwendeten Klassen enthalten. Wenn das nicht der Fall ist, waehlt LINK eine Standardreihenfolge, die nicht unbedingt mit der gewuenschten Reihenfolge uebereinstimmt. Beim Binden muss dieses Scheinprogramm die erste spezifizierte Objektdatei in der Befehlszeile von LINK sein.

Ein solches Scheinprogramm darf nicht fuer C, Pascal, FORTRAN oder BASIC verwendet werden. Diese Sprachen haben eine eigene Segmentanordnung (siehe LINK-Beschreibung). Diese Ladereihenfolge kann nicht modifiziert werden.

Mit dem /A-Schalter von MASM ist eine andere Moeglichkeit gegeben, die Segmentreihenfolge zu steuern. Dieser Schalter bewirkt, dass die Segmente in alphabetischer Reihenfolge in die Objektdatei geschrieben werden. Man muss den Segmenten Namen vergeben, die alphabetisch geordnet, die gewuenschte Ladereihenfolgé ergeben. Bei dieser Strategie ist das gleiche zu beachten wie oben, wenn das Programm aus mehreren Moduln besteht. Dann sollten alle Segmente im ersten in der LINK-Befehlszeile spezifizierten Modul angelegt sein. Weitere Informationen zum /A-Schalter sind im Abschnitt 10.3.1. enthalten.

3.4.4. Programmbeispiel

Der folgende Quelltext erlaeutert eine Verwendungsmoeglichkeit der align- und combine-Typen. Die Abbildung, die diesem Beispiel folgt, zeigt den Weg, wie LINK das gegebene Programm in den Speicher laden wuerde. Der combine-Typ memory wird nicht gezeigt, da er gleich dem public-Typ ist. Klassentypen wurden hier nicht verwendet. Sie sind ausfuehrlich im Abschnitt 3.4.3. und in dem Beispiel des Abschnitts 3.6. erklaert worden.

Beachte:

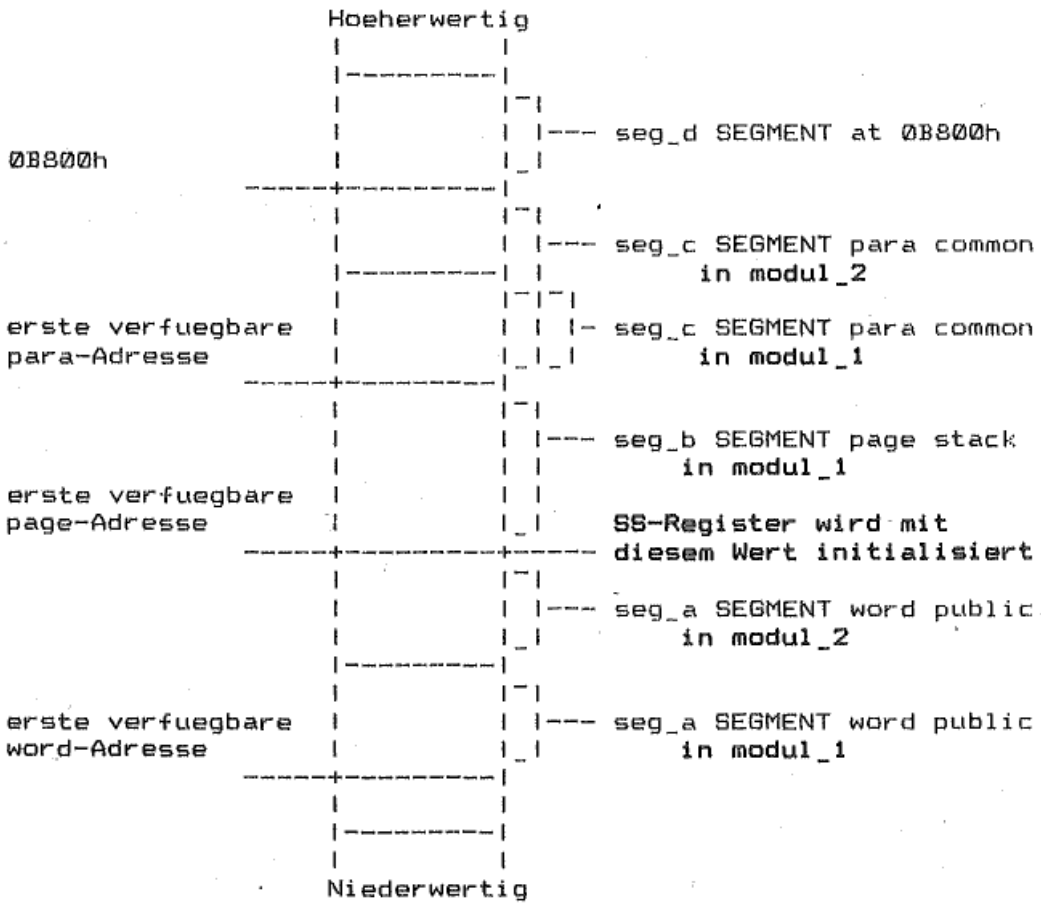
Wird ein gegebener Segmentname mehrfach in einer Quelldatei verwendet, dann muss jede dieser einen solchen identischen Namen besitzende Segmentdefinition entweder exakt die gleichen oder sich nicht wiedersprechende Attribute haben.

Beispiel:

```
            NAME    modul_1
seg_a       SEGMENT word public
beginn:       .
              .
              .
seg_a       ENDS
seg_b       SEGMENT page stack
              .
              .
              .
seg_b       ENDS
seg_c       SEGMENT para common
              .
              .
              .
seg_c       ENDS
seg_d       SEGMENT at 0B800h
              .
              .
              .
seg_d       ENDS
            END     beginn

            NAME    modul_2
seg_a       SEGMENT word public
              .
              .
              .
seg_a       ENDS
seg_b       SEGMENT page stack
              .
              .
              .
seg_b       ENDS
```

```
seg_c   SEGMENT para common
          .
          .
          .
seg_c   ENDS
        END
```



3.4.5. Segmentverschachtelung

Segmente koennen geschachtelt werden. Trifft MASM auf ein inneres Segment, unterbricht er die Uebersetzung des umschliessenden Segments und beginnt das innere Segment zu assemblieren. Nachdem das innere Segment uebersetzt ist, setzt MASM die Uebersetzung des aeusseren Segments fort. **Ein Ueberlappen der Segmente ist nicht erlaubt.**

Beispiel:

```
beisp   SEGMENT word punblic 'CODE' ;aeusseres Segment
haupt   PROC    far
          .
          .
          .
const   SEGMENT word public 'CONST' ;inneres Segment
feld    DW      feld_daten
const   ENDS                        ;Ende der
                                    ;Verschachtelung
          .
          .
          .
        RET
haupt   ENDP
beisp   ENDS
```

Dieses Programmfragment enthaelt 2 Segmente: ein Kodesegment, genannt beisp, und ein Datensegment, genannt const. Das Segment const ist vollstaendig im Segment beisp enthalten.

3.5. END

Syntax:

```
END [<ausdruck>]
```

Die END-Anweisung kennzeichnet das Ende einer Quelldatei. Der Assembler ignoriert alle dieser Pseudooperation folgenden Anweisungen.

Der wahlfreie Ausdruck definiert den Programmeintrittspunkt, die Adresse, ab der die Programmausfuehrung beginnt. Besteht das Programm aus mehreren Moduln (Quelldateien), darf nur einer die Definition eines Eintrittspunktes haben. Der Modul mit dem Eintrittspunkt ist der Hauptmodul. Wenn kein Eintrittspunkt gegeben ist, wird nichts angenommen.

Beachte:

Wurde kein Eintrittspunkt fuer das Hauptprogramm definiert, ist ein korrektes Initialisieren des Programms nicht moeglich. Das Programm wird ohne Fehler uebersetzt und gebunden. Es stuerzt aber beim Versuch der Abarbeitung ab. **Ein und nur ein Modul muss einen Eintrittspunkt haben.**

Beispiele:

```
END
END     start
```

3.6. GROUP

Syntax:

```
<name>  GROUP   <segmentname>[,<segmentname>,...]
```

Die Pseudooperation GROUP vergibt einem oder mehreren Segmenten einen Gruppennamen. Sie veranlasst, dass alle in den gegebenen Segmenten definierten Marken und variablen Adressen relativ zum Gruppenbeginn sind, also vor dem Beginn des Segments, in dem sie definiert werden.

<segmentname> muss der Name eines unter Verwendung der Pseudooperation SEGMENT oder eines SEG-Ausdruckes definierten Segments sein. (siehe dazu die Abschnitte 3.4. und 5.3.12.). Der Name darf nur einmal vorkommen.

Die Pseudooperation GROUP wirkt nicht auf die Segmentladereihenfolge. Die Ladereihenfolge haengt von der Segmentklasse oder von der Reihenfolge, in der die Objektmoduln an den Programmverbinder uebergeben werden, ab.

Weitere Erlaeuterungen dazu sind der Beschreibung des Programmverbinders zu entnehmen.

Segmente einer Gruppe muessen nicht aneinander grenzen. Nicht zu der Gruppe gehoerende Segmente koennen zwischen Segmente dieser Gruppe geladen werden. Die einzige Einschraenkung besteht darin, dass der Abstand zwischen dem 1. Byte des ersten Segments und dem letzten Byte des letzten Segments der Gruppe 65535 nicht ueberschreiten darf. Deshalb kann eine Gruppe, deren Segmente aneinander grenzen maximal 64 k Byte Speicherplatz belegen.

Gruppennamen koennen in der Pseudooperation ASSUME (Abschnitt 3.7.) **und als Segmentpraefix** (Abschnitt 5.3.7.) **verwendet werden.**

Beachte:

Ein Gruppenname darf nur in einer von mehreren moeglichen GROUP-Anweisungen in jeder Quelldatei vorkommen. Wenn verschiedene Segmente einer Quelldatei zur gleichen Gruppe gehoeren, muessen alle Segmentnamen in derselben GROUP-Anweisung angegeben werden.

Beispiel:

```
dgroup  GROUP   aseg,bseg
        ASSUME  ds:dgroup

aseg    SEGMENT byte public 'DATA1'
          .
sym_a:    .
          .
aseg    ENDS
```

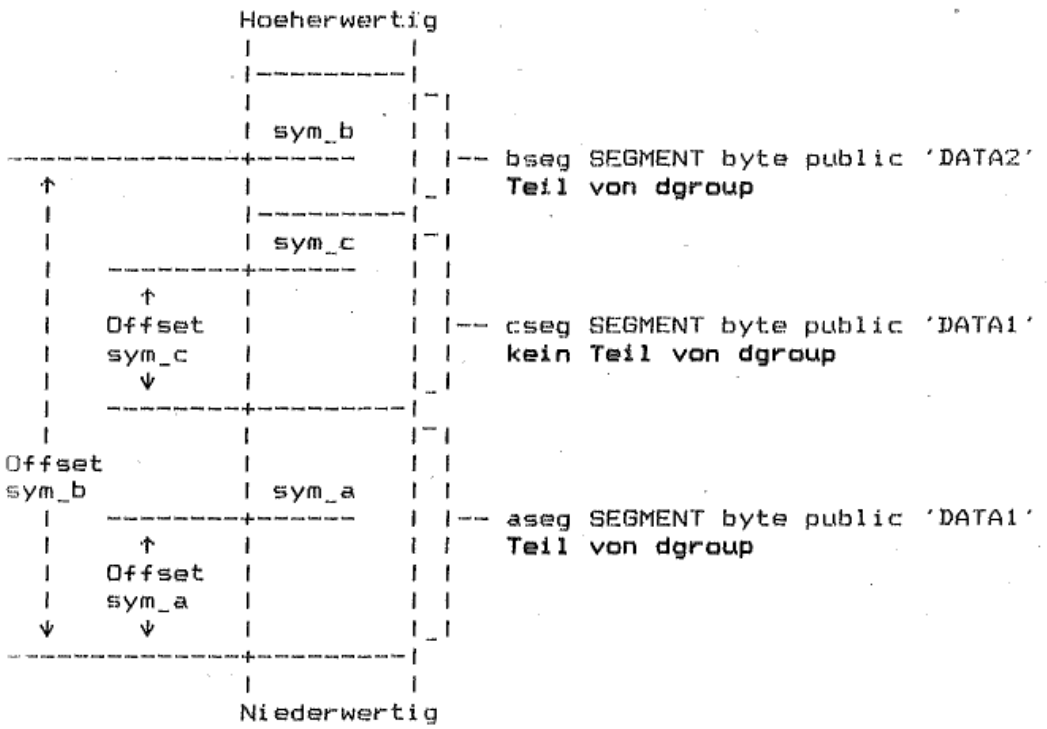
```
bseg    SEGMENT byte public 'DATA2'
          .
sym_b:    .
          .
bseg    ENDS

cseg    SEGMENT byte public 'DATA1'
          .
sym_c:    .
          .
cseg    ENDS
        END
```

Die folgende Abbildung zeigt die Reihenfolge, in der LINK die Segmente laedt.

Zuerst wird aseg geladen, weil es als erstes in der Quelldatei auftritt. Als naechstes wird cseg geladen, da es denselben Klassentyp wie aseg hat.

aseg und bseg sind erklaerte Teile derselben Gruppe, obwohl sie im Speicher getrennt stehen. Das bedeutet, die Offsets der Symbole sym_a und sym_b sind relativ zum Beginn der Gruppe, also dem Beginn von aseg. Der Offset von sym_c geht ab Beginn von cseg.



3.7. ASSUME

Syntax:

```
ASSUME <segmentregister>:<segmentname>
       [,<segmentregister>:<segmentname>,...]

ASSUME NOTHING
```

Die Pseudooperation ASSUME spezifiziert <segmentregister> als das Standardsegmentregister fuer alle Marken und Variablen, die in dem durch <segmentname> gegebenen Segment oder Gruppe, definiert sind.

Bei spaeteren Bezugnahmen auf Marken und Variablen wird automatisch das ausgewaehlte Register zur Berechnung der effektiven Adresse benutzt.

Es sind bis zu vier Angaben moeglich, fuer jedes Segmentregister eine. <segmentregister> kann einer der Segmentregisternamen CS, DS, ES oder SS sein.

<segmentname> muss einer der folgenden sein:

- Name eines vorher mit der Pseudooperation SEGMENT definierten Segments
- Name einer vorher mit der Pseudooperation GROUP definierten Gruppe
- Schluesselwort NOTHING

Das Schluesselwort NOTHING hebt die Standard-Segmentauswahl auf. Mit ASSUME NOTHING werden alle Registerzuordnungen aus einer vorangegangenen ASSUME-Anweisung aufgehoben.

Beachte:

Der Segmentpraefix kann zum Ueberschreiben des aktuellen, mit ASSUME festgelegten Segmentregisters verwendet werden.

Beispiele:

```
ASSUME  cs:CODE
ASSUME  cs:cgroup,ds:dgroup,ss:nothing,es:nothing
ASSUME  NOTHING
```

3.8. ORG

Syntax:

```
ORG <ausdruck>
```

Die Pseudooperation ORG setzt den Speicherplatzzaehler im aktuellen Segment auf <ausdruck>. Nachfolgende Befehls- und Datenadressen beginnen ab dem neuen Wert.

Das Ergebnis des Ausdrucks muss ein absoluter Wert sein. Mit anderen Worten, alle Symbole, die in <ausdruck> verwendet werden, muessen dem Assembler im Pass 1 bekannt sein.

Es kann auch das Symbol ¤ fuer den Speicherplatzzaehler verwendet werden.

Beispiele:

```
1.    ORG     120h
      mov     ax,dx
```

In diesem Beispiel beginnt die Anweisung mov ax,dx ab dem Byte 120h in dem aktuellen Segment.

```
2.            ORG     ¤+2
      feld    DW      100 DUP(0)
```

Im zweiten Beispiel beginnt die definierte Variable feld 2 Bytes nach der aktuellen Adresse.

Weitere Informationen zum Speicherplatzzaehlersymbol (¤) enthaelt der Abschnitt 5.2.4.

3.9. EVEN

Syntax:

```
      EVEN
```

Die Pseudooperation EVEN richtet das naechste Daten- oder Anweisungsbyte an einer Wortgrenze aus. Wenn der aktuelle Wert des Speicherplatzzaehlers ungerade ist, inkrementiert diese Anweisung den Zaehler, so dass ein gerader Wert entsteht und fuegt einen NOP-Befehl (keine Operation) ein.

Hat der Speicherplatzzaehler schon einen geraden Wert, so ist EVEN ohne Wirkung.

Beachte:

Diese Pseudooperation muss in Segmenten mit Byte-Ausrichtung nicht verwendet werden.

Beispiel:

```
        ORG     0
test1   DB      1
        EVEN
test2   DW      520
```

In diesem Beispiel wird durch EVEN der Speicherplatzzaehler um 1 erhoeht und ein NOP-Befehl (90h) generiert. Das bedeutet, der Offset von test2 ist 2 und nicht 1, wie er ohne diese Pseudooperation waere.

3.10. PROC und ENDP

Syntax:

```
<name>  PROC [<reichweite>]
        <anweisungen>
<name>  ENDP
```

Durch die Pseudooperationen PROC und ENDP werden Beginn und Ende einer Prozedur gekennzeichnet.

Eine Prozedur ist ein Block von Anweisungen in Form eines Unterprogramms. Jede Prozedur hat einen Aufrufnamen <name>. <name> darf nur einmal vorkommen und nicht vorher im Programm definiert worden sein.

Der wahlfreie Parameter <reichweite> kann entweder NEAR oder FAR sein. NEAR ist die Standardannahme.

<name> besitzt die gleichen Attribute wie eine Marke und kann als Operand in jump-, call- und loop-Befehlen verwendet werden.

Zwischen PROC und ENDP koennen beliebig viele Anweisungen stehen. Die Prozedur sollte als letztes eine RET-Anweisung zur Rueckkehr zum aufrufenden Punkt enthalten. Verschachtelte Prozeduren sind erlaubt.

Beispiel:

```
        push    ax              ;push 3. Parameter
        push    bx              ;push 2. Parameter
        push    cx              ;push 1. Parameter
        call    addup           ;Prozedur addup aufrufen
         .
         .
         .
```

```
addup   PROC    near            ;Rueckkehradr. fuer near
                                ;Aufruf nimmt 2 Bytes ein
        push    bp              ;Retten bp - 2 Bytes
                                ;--> Parameter beginnen
                                ; ab 4. Byte
        mov     bp,sp           ; Laden sp nach bp
        mov     ax,[bp+4]       ;1. Parameter = 4. Byte
                                ; nach Zeiger
        add     ax,[bp+6]       ;2. Parameter = 6. Byte
                                ; nach Zeiger
        add     ax,[bp+8]       ;3. Parameter = 8. Byte
                                ; nach Zeiger
        pop     bp              ;Wiederherstellen bp
        RET                     ;Rueckkehr
addup   ENDP
```

In diesem Beispiel werden 3 Zahlen als Parameter an die Prozedur addup uebergeben. Parameter werden oft durch push-Befehle vor dem Aufruf an die Prozeduren uebergeben, so dass die aufgerufene Prozedur diese aus dem Stack lesen kann.

Beachte:

Die Methode der Parameteruebergabe in diesem Beispiel stimmt mit dem in hoeheren Programmiersprachen verwendeten Standard ueberein.

Diese Prozedur koennte mit dem Stack-Trace-Kommando (K) des symbolischen Debuggers (SYMDEB) verfolgt werden.

4. Typen und Vereinbarungen

4.1. Einleitung

Dieses Kapitel erklaert:

- wie Daten fuer ein Programm generiert werden
- wie Marken und Variable, die auf Befehls- und Datenanweisungen verweisen, vereinbart werden
- wie Datentypen definiert werden, die zum Generieren von verschiedene Felder enthaltenden Datenbloecken (wie Strukturen und Records) verwendet werden koennen

4.2. Markenvereinbarungen

Eine Marke ist ein Name, der fuer die Adresse einer Anweisung steht. Marken koennen in jump-, call- und loop-Befehlen verwendet werden, um die Programmausfuehrung mit dem Befehl fortzusetzen, dessen Adresse die Marke darstellt.

4.2.1. Definition einer NEAR-Marke

Syntax:

<name>:

Diese Definition erzeugt eine Anweisungsmarke vom Typ NEAR. Die Marke kann in Anweisungen in demselben Segment verwendet werden, um die Programmsteuerung an die durch diese Marke gekennzeichnete Anweisung zu uebergeben.

<name> darf nur einmal vorkommen, nicht vorher definiert sein und muss nachfolgend mit einem **Doppelpunkt** (:) versehen sein. Spaetestens das Segment, das die Vereinbarung enthaelt, muss mit dem Segmentregister CS verbunden sein (siehe Abschnitt 3.7.). Der Assembler ordnet dem Namen den aktuellen Wert des Speicherplatzzaehlers zu.

Eine NEAR-Marke kann entweder auf einer Zeile allein oder mit einer Anweisung stehen.

Marken muessen mit PUBLIC oder EXTERN erklaert werden, wenn sie in einem Modul angelegt, und in einem anderen Modul benutzt werden (siehe Kapitel 6).

Beispiele:

```
beginn:
zyklus: inc     si
```

4.2.2. Prozedurmarken

Syntax:

<name> **PROC** [<reichweite>]

Die Pseudooperation PROC erzeugt eine Marke <name> und weist ihr wahlfrei eine <reichweite> zu. <reichweite> kann nur NEAR oder FAR sein, wobei NEAR der Standard ist. Die Marke adressiert den ersten Befehl der Prozedur.

Die Marke kann in einem call-Befehl stehen (oder in jump- oder loop-Befehlen), um die Ausfuehrungssteuerung an den ersten Befehl der Prozedur zu uebergeben.

Kommt der Assembler an die Definition der Prozedurmarke, setzt er den Wert der Marke auf den aktuellen Speicherplatzzaehler und den Typ auf NEAR oder FAR. Fuer eine Marke vom Typ FAR wird deren Segmentwert auf den des umschliessenden Segments gesetzt.

NEAR-Marken koennen in jump-, call- oder loop-Befehlen benutzt werden, um die Programmsteuerung an jede beliebige Adresse innerhalb des aktuellen Segments zu uebergeben. FAR-Marken dienen der Uebergabe der Programmsteuerung an Adressen in beliebigen

Segmenten ausserhalb des aktuellen Segments.

Prozedurmarken, die in einem Modul angelegt sind und in einem anderen aufgerufen werden, muessen mit PUBLIC und EXTRN vereinbart werden (siehe Kapitel 6).

4.3. Datenvereinbarungen

Mit Hilfe der Pseudooperationen zur Datenvereinbarung werden Daten fuer ein Programm generiert. Diese Pseudooperationen uebersetzen Zahlen, Zeichenketten und Ausdruecke in einzelne Bytes, Woerter oder andere Dateneinheiten. Die verschluesselten Daten werden in die Objektdatei uebernommen.

Folgende Pseudooperationen zur Datenvereinbarung existieren:

Pseudo-operation	Bedeutung
DB	Definiere Byte
DW	Definiere Wort (2 Bytes)
DD	Definiere Doppelwort (4 Bytes)
DQ	Definiere 4 Worte (8 Bytes)
DT	Definiere 10 Bytes

4.3.1. DB-Anweisung

Syntax:

[<name>**] DB** <initwert>**[**,<initwert>,...**]**

Fuer jeden angegebenen <initwert> wird ein Byte des Speichers zugewiesen und mit dem <initwert> selbst belegt.

<initwert> kann

- **eine ganze Zahl**
- **eine Zeichenkettenkonstante**
- **ein DUP-Operator**
- **ein konstanter Ausdruck**
- **ein Fragezeichen (?)**

sein. Das Fragezeichen steht fuer einen undefinierten Initialisierungswert. Zwei oder mehr Initialisierungswerte muessen durch Kommas (,) getrennt werden.

<name> ist wahlfrei. Wurde <name> angegeben, wird eine Variable

vom Typ Byte erzeugt, deren Offset der Wert des aktuellen Speicherplatzzaehlers ist.

Eine Zeichenkettenkonstante kann eine beliebige Anzahl Zeichen enthalten. Sie muss aber auf eine Zeile passen. Die Zeichen werden in der angegeben Reihenfolge gespeichert, mit dem ersten Zeichen an der niedrigsten und dem letzten Zeichen an der hoechsten Adresse.

Beispiele:

```
0000  10                      ganzzahl DB 16
0001  61 62                   kette    DB 'ab'
0003  45 69 6E 67 65 62 65    meldung  DB "Eingeben Name: "
      6E 20 4E 61 6D 65 20
0010  0C                      ausdr    DB 4*3
0011  ??                      undef    DB ?
0012  01 02 03 24             mehrfach DB 1,2,3,'$'
0015  000A[                   doppeln  DB 10 DUP(?)
            ??
                           ]
001F  FF                      maxbyte  DB 255
```

4.3.2. DW-Anweisung

Syntax:

```
[<name>] DW <initwert>[,<initwert>,...]
```

Fuer jeden <initwert> weist diese Pseudooperation ein Wort (zwei Bytes) des Speichers zu und initialisiert sie mit dem angegbenen Wert.

<initwert> kann

- **eine ganze Zahl**
- **eine Ein- oder Zwei-Byte-Zeichenkettenkonstante**
- **ein DUP-Operator**
- **ein konstanter Ausdruck**
- **ein Adressausdruck**
- **ein Fragezeichen (?)**

sein. Das Fragezeichen steht fuer einen undefinierten Initialisierungswert. Zwei oder mehr Initialisierungswerte muessen durch Kommas (,) getrennt werden.

<name> ist wahlfrei. Ist er angegeben, wird eine Variable vom Typ WORD erzeugt, deren Offset der Wert des aktuellen Speicherplatzzaehlers ist.

Zeichenkettenkonstanten duerfen nicht mehr als zwei Zeichen enthalten. Das letzte oder einzige Zeichen der Kette wird an der wertniedrigsten (adresshoechsten) Stelle abgelegt. Entweder 0 oder das erste Zeichen wird in der werthoechsten Stelle plat-

ziert.

Beachte:

Die Anweisungen (1) und (2)

```
(1) DW ?
(2) DW 1 DUP(?)
```

sind **nicht identisch**! Sie erzeugen verschiedenen Objektkode. Die Anweisung (1) ist **unbestimmt**. Der Wert 0 wird in den Objektkode eingetragen. (2) ist **undefiniert**. Es wird kein Wert fuer den Objektkode generiert.

In den meisten Faellen ist dieser Unterschied unbedeutend, bei Segmenten vom Combine-Typ COMMON ist er sehr wichtig.

Enthalten zwei Moduln unterschiedliche Definitionen fuer eine Variable; eine nur mit "?" und eine mit einem expliziten Wert, dann enthaelt die ausfuehrbare Datei in **Abhaengigkeit von der Bindereihenfolge** entweder 0 oder den Wert. Enthalten die Moduln "n DUP(?)" und "n DUP(m)", wobei n und m Zahlen sind, dann wird immer **unabhaengig von der Bindereihenfolge** "n DUP(m)" generiert. In COMMON-Segmenten sollte "1 DUP(?)" statt "?" verwendet werden!

Beispiele:

```
                                code      SEGMENT byte
                                ASSUME cs:code,es:code,ds:code
0000  4158                      ganzzahl  DW 16728
0002  0061                      zeichen   DW 'a'
0004  6263                      kette     DW 'bc'
0006  000C                      ausdr     DW 4*3
0008  0004  R                   adrausdr  DW kette
000A  ????                      undef     DW ?
000C  0001 0002 0003 0024       mehrfach  DW 1,2,3,'$'
0014  000A[                     doppeln   DW 10 DUP(?)
            ????
                              ]
0028  FFFF                      maxwort   DW 65535
002A  0000  R                   feldptr1  DW feld
002C  0000  R                   feldptr2  DW offset GROUP:feld
002E                            code      ENDS

0000                            group     SEGMENT word
0000  0064[                     feld      DW 100 DUP(?)
            ????
                              ]
00C8                            group  ENDS
                                END
```

4.3.3. DD-Anweisung

Syntax:

```
[<name>] DD <initwert>[,<initwert>,...]
```

Fuer jeden <initwert> weist diese Pseudooperation ein Doppelwort (4 Bytes) des Speichers zu und belegt dieses mit dem Initialisierungswert.

<initwert> kann

- eine ganze Zahl
- eine Realzahl
- eine Ein- oder Zwei-Byte-Zeichenkettenkonstante
- eine verschluesselte Realzahl
- ein DUP-Operator
- ein konstanter Ausdruck
- ein Adressausdruck
- ein Fragezeichen (?)

sein. Das Fragezeichen steht fuer einen undefinierten Initialisierungswert. Zwei oder mehr Initialisierungswerte muessen durch Kommas (,) voneinander getrennt werden.

<name> ist wahlfrei. Ist er angegeben, wird eine Variable vom Typ DWORD erzeugt, deren Offset der Wert des aktuellen Speicherplatzzaehlers ist.

Zeichenkettenkonstanten durfen nicht mehr als zwei Zeichen besitzen. Das letzte oder einzige Zeichen wird in das erste Byte und das erste Zeichen (wenn die Kette zwei Zeichen enthaelt) wird in das naechste Byte geschrieben. Alle uebrigen Bytes sind Null.

Beispiele:

```
0000  58 41 00 00           ganzzahl DD 16728
0004  61 00 00 00           zeichen  DD 'a'
0008  63 62 00 00           kette    DD 'bc'
000C  00 00 40 81           realzahl DD 1.5
0010  00 00 00 3F           real1    DD 3E000000r
0014  0C 00 00 00           ausdr    DD 4*3
0018  000C ---- R           adrausdr DD realzahl
001C  ????????              undef    DD ?
0020  01 00 00 00 02 00 00  mehrfach DD 1,2,3,'B'
      00 03 00 00 00 24 00
      00 00
0030  000A[                 doppeln  DD 10 DUP(?)
          ????????
                         ]
0058 FF FF FF FF            maxDD    DD 4294967295
```

4.3.4. DQ-Anweisung

Syntax:

```
[<name>] DQ <initwert>[,<initwert>,...]
```

Fuer jeden <initwert> werden 8 Bytes des Speichers zugewiesen und mit dem <initwert> belegt.

<initwert> kann

- eine ganze Zahl
- eine Realzahl
- eine Ein- oder Zwei-Bytes-Zeichenkettenkonstante
- eine verschluesselte Realzahl
- ein DUP-Operator
- ein konstanter Ausdruck
- ein Fragezeichen (?)

sein. Das Fragezeichen steht fuer einen undefinierten Initialisierungswert. Zwei oder mehr Initialisierungswerte muessen durch Kommas (,) getrennt werden.

<name> ist wahlfrei. Ist er angegeben, wird eine Variable vom Typ QWORD erzeugt, deren Offset der Wert des aktuellen Speicherplatzzaehlers ist.

Zeichenkettenkonstanten duerfen nicht mehr als zwei Zeichen enthalten. Das letzte (oder einzige) Zeichen wird linksbuendig in das erste Byte und das erste Zeichen (wenn die Kette aus 2 Zeichen besteht) in das naechste Byte geschrieben. Alle uebrigen Bytes nach rechts werden mit Null belegt.

Beispiele:

```
0000  58 41 00 00 00 00 00  ganzzahl DQ 16728
      00
0008  61 00 00 00 00 00 00  zeichen  DQ 'a'
      00
0010  63 62 00 00 00 00 00  kette    DQ 'bc'
      00
0018  00 00 00 00 00 00 40  real     DQ 1.5
      81
0020  00 00 00 00 00 00 00  real1    DQ 3E00000000000000r
      2E
0028  0C 00 00 00 00 00 00  ausdr    DQ 4*3
      00
0030  ????????????????     undef    DQ ?
0038  01 00 00 00 00 00 00  mehrfach DQ 1,2,3,'A'
      00 02 00 00 00 00 00
      00 00 03 00 00 00 00
      00 00 00 24 00 00 00
      00 00 00 00
```

```
0058  000A[                    doppeln  DQ 10 DUP(?)
            ????????????????
               ??
                              ]
00A8  FF FF FF FF FF FF FF    maxDQ DQ 18446744073709551615
      FF
```

4.3.5. DT-Anweisung

Syntax:

```
[<name>] DT <initwert>[,<initwert>,...]
```

Fuer jeden Initialisierungswert werden 10 Bytes Speicher zugewiesen und mit dem <initwert> belegt.

<initwert> kann

- ein ganzzahliger Ausdruck
- eine gepackte Dezimalzahl
- eine Ein- oder Zwei-Byte-Zeichenkettenkonstante
- eine verschluesselte Realzahl
- ein DUP-Operator
- ein Fragezeichen (?)

sein. Das Fragezeichen steht fuer einen undefinierten Initialisierungswert. Zwei oder mehr Initialisierungswerte muessen durch Kommas (,) getrennt werden.

<name> ist wahlfrei. Ist er angegeben, wird eine Variable vom Typ TBYTE erzeugt, deren Offset der Wert des aktuellen Speicherplatzzaehlers ist.

Zeichenkettenkonstanten duerfen nicht mehr als zwei Zeichen enthalten. Das letzte (oder einzige) Zeichen wird linksbuendig in das erste Byte und das erste Zeichen (wenn die Kette aus zwei Zeichen besteht) in das naechste Byte geschrieben. Alle uebrigen Bytes nach rechts werden mit Null belegt.

Beachte:

Die Pseudooperation DT nimmt an, dass Konstanten mit Dezimalziffern gepackte Zahlen, keine ganzen Zahlen sind. Will man eine 10-Byte lange ganze Zahl definieren, muss das durch einen der Zahl folgenden Buchstaben spezifiziert werden: z. B. D oder d fuer dezimal oder H oder h fuer hexadezimal.

Beispiele:

```
0000  90 78 56 34 12 00 00  gepackt  DT 1234567890
      00 00 00
000A  58 41 00 00 00 00 00  ganzzahl DT 16728d
      00 00 00
```

```
     0014  61 00 00 00 00 00 00  zeichen  DT 'a'
           00 00 00
     001E  63 62 00 00 00 00 00  kette    DT 'bc'
           00 00 00
     0028  00 00 00 00 00 00 00  real      DT 1.5
           C0 FF 3F
     0032  00 00 00 00 00 00 00  real1 DT 3E00000000000000000r
           00 00 3E
     003C  ??????????????????    undef     DT ?
           ?
     0046  01 00 00 00 00 00 00  mehrfach DT 1,2,3,'8'
           00 00 00 02 00 00 00
           00 00 00 00 00 00 03
           00 00 00 00 00 00 00
           00 00 24 00 00 00 00
           00 00 00 00 00
     006E  000A[                 doppeln   DT 10 DUP(?)
                 ????????????????
                    ??????
                                 ]
(*)  0078  FF FF FF FF FF FF FF  maxDT     DT
                                    120892581961462917470617 5d
```

(*) Die Definition wurde hier aus Platzgruenden auf der naechsten Zeile fortgesetzt. Das darf in der Quelldatei nicht sein!

4.3.6 DUP-Operator

Syntax:

```
<zaehler> DUP [(<initwert>)[,(<initwert>),...]]
```

Der DUP-Operator ist ein spezieller Operator. Er kann in den Pseudooperationen zur Datenvereinbarung oder in anderen Pseudooperationen zum Spezifizieren eines mehrfachen Auftretens von einem oder mehreren Initialisierungswerten verwendet werden. <zaehler> gibt an, wieviel mal <initwert> zu definieren ist.

<initwert> kann jeder Ausdruck sein, dessen Ergebnis

- **ein ganzzahliger Wert**
- **eine Zeichenkettenkonstante**
- **ein anderer DUP-Operator**

ist.

Zwei oder mehrere Initialisierungswerte sind durch Kommas (,) zu trennen.

DUP-Operatoren koennen max. bis zu einer siebzehnfachen Tiefe verschachtelt werden. Der oder die Initialisierungswerte muessen immer in Klammern eingeschlossen werden.

Beispiele:

1. **DB 100 DUP(1)**

Dieses Beispiel generiert 100 Bytes, die mit dem Wert 1 belegt werden.

2. **DW 20 DUP(1,2,3,4)**

Hier werden 80 Worte definiert. Die ersten 4 Worte werden nacheinander mit den Initialisierungswerten 1, 2, 3 und 4 belegt. Das wird wiederholt fuer die restlichen Worte.

3. **DB 5 DUP(5 DUP(5 DUP(1)))**

Diese Anweisung reserviert 125 Bytes, die alle den Wert 1 erhalten.

4. **DD 14 DUP(?)**

Das letzte Beispiel generiert 14 Doppelworte, die nicht initialisert werden.

4.4. Symbolvereinbarungen

Durch die Pseudooperationen zur Symbolvereinbarung koennen Symbole erzeugt und verwendet werden. Ein Symbol ist der beschreibende Name, der fuer eine Zahl, einen Text, einen Befehl oder eine Adresse steht.

Symbole gestalten Programme leichter lesbar und unterstuetzen die Verwendung beschreibender Namen fuer dargestellte Werte. Ein Symbol kann ueberall dort benutzt werden, wo sein entsprechender Wert erlaubt ist.

Es gibt folgende Pseudooperationen zur Symbolvereinbarung:

Pseudo-operation	Bedeutung
=	absolute Zuweisung
EQU	Gleichsetzen absoluten Wert, Aliasname oder Textsymbol
LABEL	Erzeugen einer Anweisungs- oder Datenmarke

4.4.1 Gleichheitszeichen (=)

Syntax:

<name> = <ausdruck>

Die Pseudoóperation Gleichheitszeichen erzeugt ein absolutes Symbol durch die Zuweisung des numerischen Wertes von <ausdruck> an <name>. Ein absolutes Symbol ist ein Name, der einen 16-Bit-Wert darstellt. Fuer diese Zahl wird kein Speicher zugewiesen. Statt dessen ersetzt der Assembler jedes nachfolgende Auftreten von Name durch den Wert des Ausdruckes. Dieser Wert ist waehrend der Uebersetzung verschieblich, aber zum Zeitpunkt der Abarbeitung konstant.

<ausdruck> kann

- **eine ganze Zahl**
- **eine Ein- oder Zwei-Byte-Zeichenkettenkonstante**
- **ein konstanter Ausdruck**
- **ein Adressausdruck**

sein. Sein Wert darf 65535 nicht ueberschreiten.

<name> darf entweder nur einmal vorkommen, oder er war vorher mit der Pseudooperation Gleichheitszeichen definiert worden.

Folglich koennen absolute Symbole zu jeder Zeit wieder anders definiert werden.

Beispiele:

```
0000
= 4158          ganzzahl = 16728
= 6263          kette    = 'bc'
= 000C          ausdruck = 3*4
= 6263          adrausdr = kette
0000
```

4.4.2 EQU

Syntax:

<name> **EQU** <ausdruck>

Die Pseudooperation EQU erzeugt absolute Symbole, Aliasnamen oder Textsymbole durch die Zuweisung von <ausdruck> an <name>. Ein absolutes Symbol ist ein Name, der einen 16-Bit-Wert darstellt. Ein Aliasname vertritt ein anderes Symbol. Ein Textsymbol ist ein Name, der eine Zeichenkette oder eine andere Kombination von Zeichen vertritt. Der Assembler ersetzt jedes folgende Auftreten des Namens mit dem Text oder dem Wert des Ausdrucks, in Abhaengigkeit davon, ob der Typ des Ausdrucks gegeben ist.

<name> darf nur einmal vorkommen, nicht vorher anderweitig definiert sein.

<ausdruck> kann

- eine ganze Zahl
- eine Zeichenkettenkonstante
- eine Realzahl
- eine verschluesselte Realzahl
- eine Befehlsmnemonik
- ein konstanter Ausdruck
- ein Adressausdruck

sein. Ausdruecke, deren Ergebnis Werte zwischen 0 und 65535 ergeben, erzeugen absolute Symbole und veranlassen den Assembler, <name> durch den Wert zu ersetzen. Alle anderen Ausdruecke bewirken, dass der Name durch den Text ersetzt wird.

Die Pseudooperation EQU kann zum Erzeugen einfacher Makros verwendet werden. Es ist zu beachten, dass der Assembler den Namen durch den Text oder den Wert ersetzt, bevor er versucht, die Anweisung, die diesen Namen enthaelt, zu uebersetzen.

Durch EQU definierte Symbole duerfen nicht wieder anders definiert werden.

Beispiele:

```
0000
= 0400          k         EQU 1024            ;Ersetzt durch
                                              ;den Wert
=               pi        EQU 3.14159         ;  - " -
= 0258          matrix    EQU 20*30           ;  - " -
=               staptr    EQU [bp]            ;  - " -
=               loeax     EQU xor ax,ax       ;Ersetzt durch
                                              ;den Text
=               promt     EQU 'Type Enter'    ;  - " -
=               bptr      EQU BYTE PTR        ;  - " -
0000
```

4.4.3. LABEL

Syntax:

<name> LABEL <typ>

Die Pseudooperation LABEL erzeugt eine neue Variable oder Marke durch die Zuweisung des Wertes des aktuellen Speicherplatzzaehlers und des gegebenen Typs an den Namen.

<name> darf nur einmal vorkommen und nicht vorher anders definiert worden sein.

<typ> kann einer der folgenden sein:

- BYTE
- WORD
- DWORD
- QWORD
- TBYTE
- NEAR
- FAR

<typ> kann ausserdem der Name eines gueltigen Strukturtyps sein.

Beispiele:

```
bfeld  LABEL  BYTE
wfeld  DW     100 DUP(?)
```

In diesem Beispiel verweisen bfeld und wfeld auf die gleichen Daten. Man kann als BYTE mit bfeld und als WORD mit wfeld auf die Daten zugreifen.

4.5.Typdefinitionen

Pseudooperationen zur Typdefinition ermoeglichen das Festlegen von Datentypen die aus mehreren Elementen oder Feldern bestehende Programmvariable erzeugen. Sie fassen ein oder mehrere benannte Felder unter einem gegebenen Typnamen zusammen. Dieser Typname kann dann in einer Datenvereinbarung verwendet werden, um eine Variable des gegebenen Typs zu erzeugen.

Folgende Pseudooperationen zur Typdefinition gibt es:

Pseudooperation	Bedeutung
STRUC und ENDS	Strukturtyp
RECORD	Recordtyp

4.5.1. STRUC und ENDS

Syntax:

```
<name> STRUC
       <felddefinitionen>
<name> ENDS
```

Die Pseudooperationen STRUC und ENDS kennzeichnen Beginn und Ende der Typdefinition fuer eine Struktur. Eine Strukturtypdefinition definiert den Namen des Strukturtyps und Anzahl, Typ und Standardwerte fuer die Felder, die die Struktur enthaelt.

Die Strukturdefinition erzeugt eine Maske fuer Daten. Diese Maske wird waehrend der Uebersetzung verwendet, erzeugt aber keine Daten. Daten koennen nur durch eine Strukturvereinbarung, wie sie im Abschnitt 4.6.1. beschrieben ist, generiert werden.

<name> definiert den Namen des Strukturtyps. Er darf nur einmal auftreten. <felddefinitionen> definieren die Felder der Struktur. Es koennen beliebig viele Felder definiert werden. Die Definitionen muessen einer der folgenden Formen genuegen:

```
[<name>] DB <standardwert>[,<standardwert>,...]
[<name>] DW <standardwert>[,<standardwert>,...]
[<name>] DD <standardwert>[,<standardwert>,...]
[<name>] DQ <standardwert>[,<standardwert>,...]
[<name>] DT <standardwert>[,<standardwert>,...]
```

Der wahlfreie <name> gibt dem Feld einen Namen. DB, DW, DD, DQ, DT definieren die Groesse jedes Feldes. <standardwert> definiert den Wert, den das Feld erhaelt, falls bei der Vereinbarung der Strukturvariablen kein Initialisierungswert angegeben war. <name> darf nur einmal vorkommen. Falls angegeben, stellt er den Offset vom Beginn der Struktur bis zu dem entsprechenden Feld dar.

<standardwert> kann

- **eine Zahl**
- **ein Zeichen**
- **eine Zeichenkettenkonstante**
- **ein Symbol**

sein. Er kann auch einen DUP-Operator enthalten, um mehrfach Werte fuer ein Feld zu definieren. Wenn der Standardwert eine Zeichenkettenkonstante ist, dann hat das Feld die gleiche Anzahl Bytes wie Zeichen in der Kette sind. Zwei oder mehr Standardwerte muessen durch Kommas (,) voneinander getrennt werden.

Die Definition eines Strukturtyps darf nur Felddefinitionen und Kommentare enthalten. Andere Anweisungen sind nicht erlaubt. Deshalb koennen Strukturen nicht geschachtelt werden.

Beipiel:

```
tabelle  STRUC
         zaehler DB 10
         wert    DW 10 DUP(?)
         feld    DB 'kette'
tabelle  ENDS
```

In diesem Beispiel besitzt die Struktur tabelle die Felder zaehler, wert und feld. Das Feld zaehler ist ein mit 10 initialisierter 1-Byte-Wert. wert besteht aus 10 nicht initialisierten Worten und feld ist ein mit 'kette' belegtes 5 Bytes langes Zeichenfeld . Die Offsets sind 0 fuer zaehler, 1 fuer wert und

21 fuer feld.

4.5.2. RECORD

Syntax:

```
<recordname> RECORD <feldname>:<anzahl>[=<ausdruck>]
                    [,<feldname>:<anzahl>[=<ausdruck>],...]
```

Diese Pseudooperation definiert den Recordtyp fuer einen 8- oder 16-Bit-Record. Er kann ein oder mehrere Felder enthalten.

<recordname> ist der Name des Recordtyps, der beim Erzeugen des Records verwendet wird.

<feldname> ist der Name des Feldes. <anzahl> bezeichnet die Anzahl Bits in diesem Feld, die mit <ausdruck> als Initialisierungswert belegt werden.

Es koennen beliebig viele solche Kombinationen

```
<feldname>:<anzahl>[=<ausdruck>]
```

angegeben werden. Jede muss von ihrem Vorgaenger durch ein Komma (,) getrennt sein. Die Summe der Bits ueber alle Felder darf 16 nicht ueberschreiten.

<anzahl> muss eine Konstante zwischen 1 und 16 sein. Ist die Gesamtlaenge aller Felder groesser als 8, werden 2 Bytes, sonst nur 1 Byte fuer den Record reserviert.

Ist <ausdruck> spezifiziert worden, stellt er den Standardwert fuer das Feld dar. Ist das Feld wenigsten 7 Bits lang, kann ein ASCII-Zeichen fuer <ausdruck> geschrieben werden. Der Ausdruck darf keine Vorwaertsreferenz auf ein Symbol enthalten.

Die Felder werden in der definierten Reihenfolge in den Record geschrieben. Das erste Feld steht an der adressniedrigsten Stelle und danach nach rechts das naechste. Bei einer Gesamtlaenge des Feldes von nicht genau 8 oder 16 Bits wird der gesamte Record nach rechts verschoben, so dass das letzte Bit des letzten Feldes das niederwertigste Bit in dem Byte oder Wort wird. Nicht benutzte Bits sind dann die hoechstwertigen Bits, sie werden mit 0 belegt.

Die Pseudooperation RECORD erzeugt eine **Maske** fuer Daten. Sie wird vom Assembler waehrend der Uebersetzung verwendet, generiert aber keine Daten. Daten koennen nur durch die im Abschnitt 4.6.2. beschriebene Vereinbarung eines Records erzeugt werden.

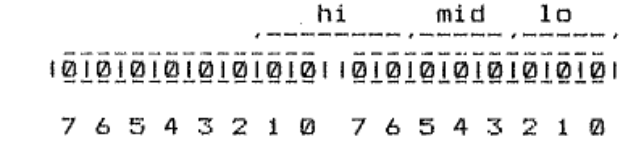
Beispiele:

1. `maske1 RECORD hi:4,mid:3,lo:3`

Das Beispiel erzeugt einen mit maske1 benannten Recordtyp, der die drei Felder hi, mid und lo enthaelt. Jede Recordvereinbarung, die diesen Typ verwendet, belegt 16 Bits.

Das Feld hi belegt die Bits 6 bis 9 (Bit 9 entspricht dem Bit 1 im hoeherwertigen Byte), das Feld mid nimmt die Bits 3 bis 5 ein und das Feld lo liegt auf den Bits 0 bis 2. Alle uebrigen Bits werden nicht benutzt.

Das folgende Diagramm zeigt diesen Recordtyp:

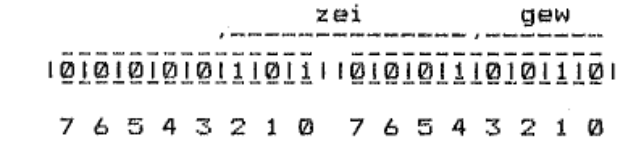


Da keine Initialisierungswerte angegeben wurden, erhalten alle Bits den Wert 0. **Es ist zu beachten, dass das nur eine vom Assembler unterstuetzte Maske ist, Daten werden nicht erzeugt.**

2. **maske2 RECORD zei:7='Q',gew:4=2**

Hier wird ein Recordtyp maske2 mit zwei Feldern, zei und gew erzeugt. Sie werden mit dem Buchstaben Q und der Zahl 2 initialisiert. Nicht benutzte Bits werden 0 gesetzt.

Das folgende Diagramm zeigt diesen Recordtyp:



4.6. Struktur- und Recordvereinbarung

Diese Pseudooperationen generieren aus mehreren Elementen oder Feldern bestehende Datenbytebloecke. Die Vereinbarung einer Struktur oder eines Records besteht aus dem Namen einer vorher erfolgten Definition eines Struktur- bzw. Recordtyps und einer Menge von Initialisierungswerten.

4.6.1. Strukturvereinbarung

Syntax:

[<name>] <strukturname> <[<initwert>[,<initwert>,...]]>

Eine Strukturvariable ist eine Variable mit ein oder mehreren Feldern verschiedener Laenge.

<name> ist der Name der Variablen. <strukturname> ist der Name eines mittels der Pseudooperation STRUC definierten Struktur-

typs. Fuer jedes Feld kann ein <initwert> angegeben werden. Sie stellen die Initialisierungswerte fuer die Struktur dar.

<name> ist wahlfrei. Ist er nicht angegeben, weist der Assembler Speicherplatz fuer die Struktur zu, aber er erzeugt keinen Namen, mit dem auf die Struktur zugegriffen werden kann.

<initwert> kann

- **eine ganze Zahl**
- **eine Zeichenkettenkonstante**
- **ein Ausdruck**

sein. Die spitzen Klammern (<>) muessen auch dann geschrieben werden, wenn kein Initialisierungswert angegeben wird. Zwei oder mehr Initialisierungswerte muessen durch Kommas (,) voneinander getrennt werden. Bei der Verwendung des DUP-Operators brauchen nur die Werte innerhalb der runden Klammern in spitze Klammern eingeschloessen zu werden (siehe Abschnitt 4.3.6.).

Es muessen nicht alle Felder einer Struktur initialisiert werden. Wurde ein Initialisierungswert weggelassen, setzt der Assembler automatisch den Standardwert aus dem Strukturtyp ein. Ist der auch nicht vorhanden, bleibt das Feld uninitialisiert.

Im Abschnitt 5.2.9. werden verschiedene Moeglichkeiten fuer die Verwendung von Strukturdaten gezeigt.

Beachte:

Die bei Definition des Strukturtyps festgelegten Standardwerte koennen nicht in jedem Fall durch den Initialisierungswert bei der Strukturvereinbarung ueberschrieben werden. Das ist nur moeglich, wenn kein Mehrfachwert vorliegt.

Es sei beispielsweise folgende Definition eines Strukturtyps angenommen:

```
kette   STRUC
        puffer  DB   100 DUP(?)     ;kann nicht
                                    ;ueberschriben werden
        crlf    DB   13,10          ;  - " -
        frage   DB   'Dateiname:    ;kann ueberschrieben
                                    ;werden
        endekz  DB   36             ;  - " -
kette   ENDS
```

Die Variablen puffer und crlf koennen bei der Strukturvereinbarung nicht ueberschrieben werden, weil sie Mehrfachwerte als Standardwerte haben. Die Variable frage kann in ihrer gesamten Laenge ueberschrieben werden (nicht mehr als 11 Bytes). Analog kann das Feld endekz mit jedem Bytewert ueberschrieben werden.

Beispiele:

1. **struc1 tabelle <>**

Das Beispiel erzeugt eine Strukturvariable struc1, deren Typ durch den Strukturtyp tabelle gegeben ist.

Die Initialisierungswerte fuer die Felder der Struktur werden durch die moeglicherweise angegebenen Standardwerte des Strukturtyps ersetzt. Hier koennte beispielsweise die Definition des Strukturtyps tabelle aus dem Beispiel des Abschnitts 4.5.1. verwendet werden.

Das Ergebnis waere: Das erste Byte besitzt den Wert 10, danach folgen 10 uninitialisierte Worte und anschliessend die Zeichenkette 'kette'.

2. **struc2 tabelle <0,,>**

Das zweite Beispiel erzeugt eine Strukturvariable struc2. Ihr Typ soll auch tabelle sein. Der Initialisierungswert fuer das erste Feld wird auf 0 gesetzt. Die durch den Strukturtyp definierten Standardwerte werden fuer die restlichen Felder verwendet.

Nehmen wir an, es wuerde wieder der Strukturtyp aus dem Abschnitt 4.5.1. benutzt. Der Initialisierungswert 0 aus der Strukturvereinbarung ueberschreibt dann den Standardwert 10 aus der Definition des Strukturtyps.

3. **struc3 tabelle 10 DUP(<0,,>)**

Das letzte Beispiel erzeugt eine Variable, die 10 Strukturen des Typs tabelle enthaelt. Das erste Feld jeder Struktur wird auf den Initialisierungswert 0 gesetzt, die uebrigen Felder erhalten den Standardwert.

4.6.2. Recordvereinbarung

Syntax:

[<name>] <recordname> <[<initwert>[,<initwert>,...]]>

Eine Recordvariable ist ein 8- oder 16-Bitwert, deren Bits auf ein oder mehrere Felder aufgeteilt sind.

<name> ist der Name der Variablen. <recordname> ist der Name eines mittels der Pseudooperation RECORD definierten Recordtyps. <initwert> definiert den Initialisierungswert fuer ein Feld des Records.

Der Name ist wahlfrei. Wurde kein Name angegeben, weist der Assembler Speicherplatz fuer den Record zu, aber er erzeugt keine Variable, mit der auf den Record zugegriffen werden kann.

<initwert> ist wahlfrei. Er kann eine ganze Zahl, eine Zeichenkettenkonstante oder jeder Ausdruck, dessen Ergebniswert nicht groesser als die aus dem Recordtyp resultierende Bitzahl ist, sein. Die spitzen Klammern (<>) sind auch dann erforderlich, wenn kein Initialisierungswert angegeben wird. Mehrere Initialisierungswerte sind durch Kommas (,) voneinander zu trennen. Wird der DUP-Operator (siehe Abschnitt 4.3.6.) verwendet, brauchen nur die Werte innerhalb der runden KLammern in spitze Klammern eingeschlossen werden.

Es muessen nicht alle Felder eines Records initialisiert werden. Wird ein Initialisierungswert weggelassen, verwendet der Assembler automatisch den Standardwert fuer dieses Feld. Dieser wird durch den Recordtyp definiert. Ist auch dieser nicht vorhanden, dann ist das Feld uninitialisiert.

Beispiele:

1. **rec1 maske1 <>**

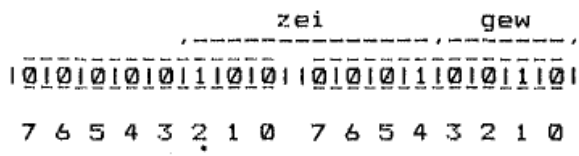
Das erste Beispiel erzeugt eine Variable mit dem Namen rec1, deren Typ durch den Recordtyp maske1 gegeben ist.

Die Initialisierungswerte fuer die Felder des Records werden durch die moeglicherweise angegebenen Standardwerte des Recordtyps gesetzt.

Hier koennte beispielsweise die Definition des Recordtyps maske1 aus dem Abschnitt 4.5.2. verwendet werden. Dann wuerde rec1 0 sein, weil alle Felder in der Definition nicht initialisiert waren.

2. **tab maske2 10 DUP(<'A',2>)**

Dieses Beispiel erzeugt eine Variable mit dem Namen tab, die 10 Records vom Recordtyp maske2 enthaelt. Die Felder in diesem Record werden alle auf die Werte A und 2 gesetzt. Sei maske2 der Recordtyp aus dem Beispiel im Abschnitt 4.5.2., dann ueberschreibt A den Standardwert 0 in der Definition des Recordtyps.



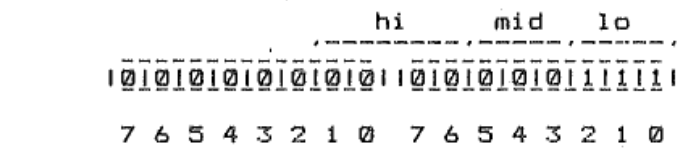
Dieses Bitdiagramm zeigt einen von 10 Records, die durch diese Vereinbarung generiert werden.

3. **para maske1 <,,7>**

Das letzte Beispiel erzeugt eine Recordvariable mit dem Namen para. Ihr Typ ist maske1. Die Initialisierungswerte fuer die ersten beiden Felder sind die in der Definition des Recordtyps angegebenen Standardwerte. Der Wert fuer das 3. Feld ist 7. Wenn

die Definition des Recordtyps aus dem Abschnitt 4.5.2. verwendet wird, erhalten die ersten beiden Felder die Werte 0, weil sie nicht initialisiert waren.

Das folgende Bitdiagramm zeigt das Ergebnis:



5. Operanden und Ausdruecke

5.1. Einleitung

Dieses Kapitel beschreibt die Syntax und die Bedeutung von Operanden und Ausdruecken, die in Assemblerbefehlen und Pseudooperationen verwendet werden.

Operanden stellen Werte, Register oder Speicherplatzzaehler dar, die in Maschinenbefehlen und Pseudooperationen angewendet werden.

Ausdruecke sind Kombinationen von Operanden mit arithmetischen, logischen, Bit- und Attributoperatoren. Sie dienen der Berechnung von Werten oder Speicherplatzzaehlern, die in den Befehlen und Pseudooperationen bestimmte Wirkungen hervorrufen.

Operatoren zeigen an, welche Operationen mit einem oder mehreren Werten in einem Ausdruck zur Berechnung seines Wertes ausgefuehrt werden sollen.

5.2. Operanden

Ein Operand kann eine Konstante, eine Marke, eine Variable oder ein anderes Symbol sein. Ein Operand wird in Befehlen und Pseudooperationen verwendet und stellt einen Wert, ein Register oder einen Speicherplatzzaehler dar.

Folgende Operandentypen existieren:

- Konstante
- Direktadresse
- verschiebliche Operanden
- Speicherplatzzaehleroperand
- Registeroperand
- Basisoperanden
- Indexoperanden
- Basis-Index-Operanden
- Strukturoperanden
- Recordoperanden
- Recordfeldoperanden

5.2.1. Konstanten

Syntax:

{<zahl> | <zeichenkette> | <ausdruck>}

Eine Konstante ist eine Zahl, ein Symbol oder ein Ausdruck, dessen Ergebnis ein fester Wert ist. Im Gegensatz zu anderen Operanden sind Konstanten waehrend der Abarbeitung unveraenderlich.

Beispiele:

```
mov     ax,9
mov     al,'c'
mov     bx,65535/3
mov     cx,zaehler
```

Man beachte, dass zaehler im letzten Beispiel nur dann eine Konstante ist, wenn sie mit EQU oder dem Operator = definiert worden ist. Ist zaehler ein Symbol, ein verschieblicher Wert oder eine Adresse, dann ist er keine Konstante.

5.2.2. Direktadresse

Syntax:

:<offset>

Eine Direktadresse ist ein aus und <offset> bestehendes Adressenpaar. Sie stellt eine absolute Speicheradresse von einem oder mehreren Speicherbytes dar.

 kann ein Segmentregister (CS, DS, SS, ES), ein Segmentname oder ein Gruppenname sein.

<offset> muss eine ganze Zahl, ein absolutes Symbol oder ein Ausdruck, dessen Ergebnis im Bereich von 0 bis 65535 liegt, sein.

Beispiele:

```
mov     ax,ss:0031h
mov     bx,data:0
mov     ax,DGROUP:block
```

5.2.3. Verschiebliche Operanden

Syntax:

<symbol>

Ein verschieblicher Operand ist ein Symbol, das die Speicheradresse (Segment und Offset) eines Befehls oder Datenbereichs darstellt. Im Gegensatz zu Direktadressen sind verschiebliche Operanden relativ zum Beginn des Segments oder der Gruppe, in dem sie definiert sind. Sie haben so lange keinen expliziten Wert, bis das Programm gebunden ist.

Beispiele:

```
call    prog
mov     bx,wert
mov     bx,OFFSET dgroup:tab
mov     cx,zaehler
```

Es ist zu beachten, dass zaehler im letzten Beispiel ein verschieblicher Operand ist, wenn er mit der Pseudooperation DW definiert wurde. Mit EQU oder = definiert, ergibt er eine Konstante.

5.2.4.Speicherplatzzaehleroperand

Syntax:

```
$
```

Der Speicherplatzaehleroperand ist ein spezieller Operand. Er repraesentiert waehrend der Uebersetzung den aktuellen Speicherplatzzaehler im aktuellen Segment. Dieser Operand weist die gleichen Attribute wie eine Marke vom Typ NEAR auf.

Er stellt eine Befehlsadresse dar, die relativ zum aktuellen Segment ist. Ihr Offset ist gleich der Anzahl Bytes, die bis zu dieser Stelle in dem Segment generiert worden sind. Nach jedem uebersetzten Befehl erhoeht der Assembler den Speicherplatzzaehler um die Anzahl generierter Bytes.

Beispiel:

```
menue   DB      'Programmfunktionen: ',13,10
f1      DB      'F1 Aendern',13,10
f2      DB      'F2 Retten Datei',13,10
        .
        .
        .
f10     DB      'Programmende',13,10,'$'
laenge  =       $-menue
```

In diesem Beispiel wird der Speicherplatzzaehleroperand benutzt, um die Gesamtlaenge dieser Zeichenkettenvereinbarungen festzustellen.

5.2.5. Registeroperand

Syntax:

<registername>

Ein Registeroperand ist der Name eines CPU-Register. Die Befehle fuehren mit dem Inhalt der in den Registeroperanden angegebenen Registern Aktionen durch. <registername> kann jeder in der folgenden Tabelle aufgefuehrte Registername sein.

Registeroperandentyp	Registernamen			
allgemeines 16-Bit-Register	AX	BX	CX	DX
8-Bit-Hoeherwertiges-Register	AH	BH	CH	DH
8-Bit-Niederwertiges-Register	AL	BL	CL	DL
16-Bit-Segmentregister	CS	DS	SS	ES
16-Bit-Zeiger-/Index-Register	SP	BP	SI	DI

Es ist jede moegliche Schreibweise von kleinen und grossen Buchstaben erlaubt.

AX, BX, CX, DX sind allgemeine 16-Bit-Register. Sie koennen fuer alle Daten- oder numerischen Manipulationen verwendet werden. AH, BH, CH, DH sind die hoeherwertigen 8 Bits dieser allgemeinen Register. Entsprechend sind AL, BL, CL, DL die niederwertigen 8 Bits.

CS, DS, SS, ES sind Segmentregister. Sie enthalten die aktuellen Segmentadressen fuer das Kodesegment, das Datensegment, das Stacksegment und das Extrasegment. Alle Befehls- oder Datenadressen sind relativ zur Segmentadresse in einem dieser Register.

SP ist das 16-Bit-Register fuer den Stackpointer. Der Stackpointer enthaelt die aktuelle hoechste Stackadresse. Diese Adresse ist relativ zur Segmentadresse im SS-Register. Sie wird automatisch bei Stackbefehlen veraendert.

BX, BP, DI, SI sind 16-Bit-Basis- bzw. Index-Register. Diese allgemeinen Register werden ueblicherweise als Zeiger auf Programmdaten verwendet. Adressausdruecke, die das BP-Register enthalten, haben standardmaessig einen Offset zu dem durch das SS-Register adressierte Segment. Adressausdruecke, in denen BX, SI oder DI verwendet werden, haben einen Offset in das durch das DS-Register adressierte Segment. Wird das DI-Register mit Zeichenketten-Befehlen verwendet, stellt es immer einen Offset zum ES-Segment dar.

Das unbenannte 16-Bit-Register enthaelt 9 1-Bit-Flags, deren Positionen und Bedeutung in der folgenden Tabelle erklaert sind.

Flagbit	Bedeutung
0	CF Uebertragsflag (Carryflag)
2	PF Paritaetsflag
4	AF Hilfsuebertragsflag (Auxiliary flag)
6	ZF Nullflag (Zero flag)
7	SF Vorzeichenflag
8	TF Einzelschrittflag - Unterbrechnung nach Befehlsausfuehrung (Trap)
9	IF Interruptflag
10	DF Richtungsflag
11	OF Ueberlaufflag

Obwohl dieses 16-Bit-Register keinen Namen hat, kann auf den Inhalt des Registers durch die Befehle LAHF, SAHF, PUSHF und POPF zugegriffen werden. Weiter Erlaeuterungen zu den Flags sind in der Befehlsbeschreibung enthalten.

5.2.6. Basisoperand

Syntax:

```
<verschiebung> [BP]
<verschiebung> [BX]
```

Ein Basisoperand stellt eine Speicheradresse relativ zu einem der Basisregister BP oder BX dar.

<verschiebung> kann

- **ein Direktwert**
- **eine Direktadresse**

sein. Sie muss einen absoluten Wert oder eine Speicheradresse ergeben. Wurde keine Verschiebung angegeben, wird Null angenommen. Die effektive Adresse eines Basisoperanden berechnet sich aus der Summe von Verschiebung und Inhalt des gegebenen Registers.

Wird BP verwendet, dann ist die Operandenadresse relativ zu dem

Segment, auf das das SS-Register zeigt.

Bei Verwendung von BX ist die Adresse relativ zu dem Segment, auf das das DS-Register zeigt.

Basisoperanden koennen unterschiedliche Formen haben. Die folgenden Schreibweisen sind aequivalent:

```
[<verschiebung>][BP]
[BP+<verschiebung>]
[BP].<verschiebung>
[BP]+<verschiebung>
```

In jedem Fall ergibt sich die effektive Adresse aus der Summe von Verschiebung und Inhalt des angegebenen Registers.

Beachte:

Die eckigen Klammern im Adressfeld zeigen nur dann eine Speicherreferenz an, wenn wenigstens eines der Register BX, BP, SI oder DI in den eckigen Klammern enthalten ist. Zum Beispiel transportiert mov ax,[BX] das Wort, auf welches BX weist, nach AX. Aber mov ax,[2] traegt den Direktwert 2 in das Register AX ein. Eine Speicherreferenz als absoluter Offset muesste so generiert werden: mov ax,ds:[2]. Weil DS das Standardsegmentregister fuer eine Speicherreferenz ist, wird der Segmentpraefix nicht in die Objektdatei geschrieben. Es wird dem Assembler aber mitgeteilt, dass [2] eine Speicherreferenz und kein Direktwert ist.

Beispiele:

```
mov     ax,[bp]
mov     ax,[bx]
mov     ax,12[bx]
mov     ax,tab[bp]
```

5.2.7. Indexoperand

Syntax:

```
<verschiebung>[SI]
<verschiebung>[DI]
```

Ein Indexoperand stellt eine Speicheradresse relativ zu einem der Indexregister SI oder DI dar.

<verschiebung> kann

- **ein Direktwert**
- **eine Direktadresse**

sein. Sie muss einen absoluten Wert oder eine Speicheradresse ergeben. Es wird Null angenommen, wenn keine Verschiebung spezi-

fiziert wird.

Die effektive Adresse eines Indexoperanden ist die Summe aus der Verschiebung und dem Inhalt des angegebenen Registers. Die Adresse ist relativ zu dem Segment, auf das das DS-Register zeigt.

Indexoperanden koennen unterschiedliche Formen haben. Folgende Schreibweisen sind aequivalent:

```
[<verschiebung>][DI]
[DI+<verschiebung>]
[DI].<verschiebung>
[DI]+<verschiebung>
```

In jedem Fall ergibt sich die effektive Adresse aus der Summe von Verschiebung und Inhalt des angegebenen Registers.

Beachte:

Die eckigen Klammern im Adressfeld zeigen nur dann eine Speicherreferenz an, wenn wenigstens eines der Register BX, BP, SI oder DI in den eckigen Klammern enthalten ist. Zum Beispiel transportiert mov ax,[BX] das Wort, auf welches BX weist, nach AX. Aber mov ax,[2] bringt den Direktwert 2 in das Register AX. Eine Speicherreferenz als absoluter Offset muesste so generiert werden: mov ax,ds:[2]. Weil DS das Standardsegmentregister fuer eine Speicherreferenz ist, wird der Segmentpraefix nicht in die Objektdatei geschrieben. Es wird dem Assembler aber mitgeteilt, dass [2] eine Speicherreferenz und kein Direktwert ist.

Beispiele:

```
mov     ax,[si]
mov     ax,[di]
mov     ax,12[di]
mov     ax,tab[si]
```

5.2.8. Basis-Index-Operand

Syntax:

```
<verschiebung>[BP][SI]
<verschiebung>[BP][DI]
<verschiebung>[BX][SI]
<verschiebung>[BX][DI]
```

Ein Basis-Index-Operand stellt eine Speicheradresse relativ zu einer Kombination aus Basis- und Indexregister dar.

<verschiebung> kann

- **ein Direktwert**
- **eine Direktadresse**

sein. Das Ergebnis muss eine absolute Zahl oder eine Speicheradresse ergeben. Es wird Null angenommen, wenn keine Verschiebung spezifiziert war.

Die effektive Adresse eines Basis-Index-Operanden errechnet sich aus der Summe von Verschiebung und dem Inhalt der beiden angegebenen Register. Bei Verwendung des BP-Registers ist die Adresse relativ zu dem Segment, auf das das SS-Register zeigt. Sonst ist sie relativ zu dem Segment, auf das das DS-Register weist.

Basis-Index-Operanden koennen unterschiedliche Formen haben. Folgende Schreibweisen sind aequivalent:

```
[<verschiebung>][BP][DI]
[BP+DI]+<verschiebung>
[BP+DI].<verschiebung>
[DI]+<verschiebung>+[BP]
```

In jedem Fall ergibt sich die effektive Adresse aus der Summe von Verschiebung und dem Inhalt der beiden angegebenen Register.

Jedes Basisregister kann mit jedem Indexregister kombiniert werden, aber es ist nicht erlaubt, zwei Basis- oder zwei Indexregister zu kombinieren.

Beispiele:

```
mov     ax,[bp][si]
mov     ax,[bx+di]
mov     ax,12[bp+di]
mov     ax,tab[bx][si]
mov     ax,tab[bx][bp]  ;Fehler - 2 Basisregister
mov     ax,tab[di][si]  ;Fehler - 2 Indexregister
```

5.2.9. Strukturoperand

Syntax:

```
<variable>.<feld>
```

Ein Strukturoperand ist die Speicheradresse des Feldes einer Struktur. <variable> muss entweder ein Strukturname sein oder ein Speicheroperand, der die Adresse einer Struktur ergibt. <feld> muss der Name eines Feldes innerhalb einer Struktur sein. <variable> und <feld> sind durch den Struktur-Feldname-Operator Punkt (.) voneinander zu trennen. Dieser Operator ist im Abschnitt 5.3.8. beschrieben.

Die effektive Adresse eines Strukturoperanden ergibt sich aus

der Summe der Offsets von <variable> und <feld>. Die Adresse ist relativ zu dem Segment oder der Gruppe, in welcher <variable> definiert wurde.

Beispiele:

```
1.    datum       STRUC
                  tag       DW      ?
                  monat     DW      ?
                  jahr      DW      ?
      datum       ENDS

      aktuelles_datum datum   <'01','ja','86'>

                  mov       ax,aktuelles_datum.tag
                  mov       aktuelles_datum.jahr,'87'
```

In diesem Beispiel ist die Struktur erst vereinbart und dann definiert worden. Der erste MOV-Befehl bringt '01' (den Wert von aktuelles_datum.tag) in das Register AX. Der naechste Befehl legt den Wert '87' in die Variable aktuelles_datum.jahr ab.

```
2.    stackrahmen       STRUC
            retadr      DW      ?       ;Stackrahmen
            ziel        DW      ?       ;von der niedrigsten ...
            quelle      DW      ?
            nbytes      DW      ?       ;... zur hoechsten Adr.
      stackrahmen       ENDS

      copy      PROC    NEAR    ;Retten von nbytes, quelle und
                                ;ziel, bevor die Prozedur aufge-
                                ;rufen wird
                mov     bx,sp   ;Laden Stack in Basisregister
                mov     ax,ds
                mov     es,ax                   ;(es) = Datensegment
                mov     di,ss:[bx].ziel         ;(di) = Ziel
                mov     si,ss:[bx].quelle       ;(si) = quelle
                mov     cx,ss:[bx].nbytes       ;(cx) = nbytes
                rep     movsb   ;Transport Daten ds:si -> es:di
                ret
      copy      ENDP
```

In diesem Beispiel werden die Strukturoperanden zum Zugriff auf die Werte im Stack benutzt.

Beachte:

Die Prozedur im obigen Beispiel entspricht nicht den Methoden der Parameteruebergabe bei hoeheren Programmiersprachen! Fuer diese Beispielprozedur kann das Stack-Trace-Kommando (K) im SYMDEB nicht benutzt werden.

5.2.10. Recordoperand

Syntax:

```
<recordname> [<wert>[,<wert>,...]]
```

Ein Record-Operand verweist auf den Wert eines Recordtyps. Diese Operanden koennen in Ausdruecken verwendet werden.

<recordname> muss der Name eines in dieser Quelldatei definierten Recordtyps sein. Der wahlfreie <wert> ist der Wert eines Feldes im Record. Wird mehr als ein Wert angegeben, so sind sie durch Kommas (,) voneinander zu trennen. Werte koennen Ausdruecke oder Symbole sein, wenn sie eine Konstante ergeben.

Die umschliessenden spitzen Klammen (<>) muessen geschrieben werden, auch wenn keine Werte eingetragen sind. Wird fuer ein Feld kein Wert angegeben, wird der Standardwert fuer dieses Feld eingesetzt.

In dem folgenden Beispiel wird diese Recorddefinition angenommen:

```
maske1 RECORD hi:4,mid:3,lo:3
```

Beispiel:

```
rec1    maske1  <3,2,1>
        mov     ax,rec1
```

In diesem Beispiel wird der konstante Wert 209 (0D1h) in das Register AX transportiert.

Das folgende Bitdiagramm zeigt diesen Wert:

```
                 hi      mid   lo
              ,--------,-----,-----,
 ---------------  ---------------
|0|0|0|0|0|0|0|0||1|1|0|1|0|0|0|1|

 7 6 5 4 3 2 1 0  7 6 5 4 3 2 1 0
```

Die Verwendung von Record-Operanden ist aehnlich der Vereinbarung von Records mit der Ausnahme, dass hier konstante Daten verwendet werden. Siehe dazu auch den Abschnitt 4.6.2.

5.2.11. Record-Feld-Operand

Syntax:

```
<recordfeldname>
```

Der Record-Feld-Operand stellt den Platz eines Feldes im entsprechenden Record dar. Der Operand ergibt die Bitposition des Feldes im Record (den Offset) und kann wie ein konstanter Ope-

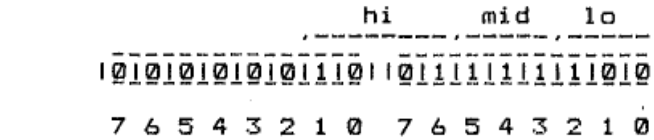
rand verwendet werden.

<recordfeldname> muss der Name eines vorher definierten Recordfeldes sein.

Im naechsten Beispiel werden die folgende Recordvereinbarung und -definition angenommen:

```
maske1  RECORD  hi:4,mid:3,lo:3
rec1    maske1  <9,7,4>
```

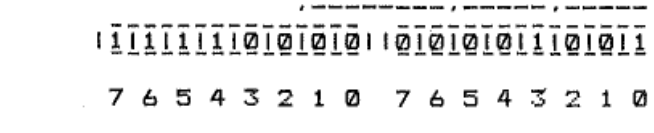
rec1 hat jetzt den Wert 636 (27Ch), dargestellt im folgenden Bitdiagramm:



Beispiel:

```
mov     cl,hi
mov     dx,rec1
ror     dx,cl
mov     rec1,dx
```

In diesem Beispiel wird eine 6, der Verschiebefaktor hi (der Offset des Feldes hi im Record) in das Register CL gebracht. Der Inhalt von rec1 kommt ins Register DX. Der Offset des 3. Feldes (hi) wird fuer die Rotation des Wertes von rec1 benutzt, so dass der Wert von hi jetzt auf den niederwertigsten Bits steht. Dieser neue Wert wird in rec1 abgelegt. rec1 besitzt den Wert 61449 (0F009h). Er ist im folgenden Bitdiagramm dargestellt.



5.3. Operatoren und Ausdruecke

Ein Ausdruck ist eine Kombination von Operanden und Operatoren, die einen einzigen Wert ergibt. Ausdruecke koennen fuer jeden in diesem Kapitel beschriebenen Operanden stehen. Das Ergebnis eines Ausdrucks kann ein Wert oder eine Speicheradresse sein, abhaengig vom Typ der verwendeten Operanden und Operatoren.

Der Assembler bietet eine Vielzahl von Operatoren - arithmetische, Verschiebe-, Vergleichs- und Bitoperatoren - zum Veraendern und Vergleichen der Werte von Operanden. Attributoperatoren veraendern die Attribute der Operanden, sowie ihren Typ, ihre Adresse und ihre Groesse.

Zusaetzlich zu den hier beschriebenen Operatoren koennen der DUP-Operator (siehe Abschnitt 4.3.6.) und die speziellen Makrooperatoren (siehe Abschnitt 8.3.) verwendet werden.

5.3.1. Arithmetische Operatoren

Syntax:

```
<ausdruck1> * <ausdruck2>
<ausdruck1> / <ausdruck2>
<ausdruck1> MOD <ausdruck2>
<ausdruck1> + <ausdruck2>
<ausdruck1> - <ausdruck2>
+ <ausdruck>
- <ausdruck>
```

Arithmetische Operatoren ermoeglichen die allgemeinen mathematischen Operationen. In der folgenden Tabelle sind diese Operatoren und ihre Bedeutung aufgelistet:

Operator	Bedeutung
+	Positives Vorzeichen
-	Negatives Vorzeichen
*	Multiplikation
/	Ganzzahlige Division
MOD	Rest nach der Division (Modulo)
+	Addition
-	Subtraktion

Bei allen arithmetischen Operatoren, mit Ausnahme von + und -, muessen <ausdruck1> und <ausdruck2> ganze Zahlen sein. Der Operator + kann verwendet werden, um eine ganze Zahl zu einem verschieblichen Speicheroperanden zu addieren. Mit dem Operator - kann von einem verschieblichen Speicheroperanden eine ganze Zahl subtrahiert werden. Er kann auch zum Subtrahieren eines verschieblichen Operanden von einem anderen verschieblichen Operanden verwendet werden. Dies ist aber nur moeglich, wenn die Operanden auf Plaetze desselben Segments verweisen. Das Ergebnis ist ein absoluter Wert.

Beachte:

Die Vorzeichen + und - (verwendet zum Kennzeichen von positiven und negativen Zahlen) sind nicht identisch mit den Binaeroperatoren + und - (verwendet zum Bezeichnen von Addition und Subtraktion). Die Vorzeichen + und - besitzen eine hoehere Prioritaet (siehe Tabelle im Abschnitt 5.4.).

Beispiele:

```
14*4            ;=56
14/4            ;=3
14 MOD 4        ;=2
14+4            ;=18
14-4            ;=10
14- +4          ;=10
14- -4          ;=18
alpha+5         ;addiere 5 zu dem Offset von alpha
alpha-5         ;vom Offset von alpha 5 subtrahieren
alpha-beta      ;vom Offset von alpha den Offset von
                ; beta subtrahieren
```

5.3.2. SHR- und SHL-Operator

Syntax:

```
<ausdruck> SHR <anzahl>
<ausdruck> SHL <anzahl>
```

Bie Operatoren SHR und SHL verschieben <ausdruck> nach rechts oder links um die mit <anzahl> gegebene Anzahl Bits. Am Ende herausgeschobene Bits gehen verloren. Ist die Anzahl groesser oder gleich 16, so ist das Ergebnis der Operation Null.

Es werden 8 oder 16 Bits verschoben, abhaengig davon, ob der zu verschiebende Wert ein Wort oder ein Byte ist.

Beachte:

Man darf die Assembleroperatoren SHR und SHL nicht mit den Prozessorbefehlen gleichen Namens verwechseln.

Beispiele:

```
mov     ax,01110111b SHL 3      ;0000001110111000b -> ax
mov     ah,01110111b SHR 3      ;00001110b -> ax
```

Es ist zu beachten, dass im ersten Beispiel in dem Wortregister 16 Bits verschoben werden, im zweiten Beispiel aber nur 8, weil das Register ah nur 1 Byte fasst.

5.3.3. Vergleichsoperatoren

Syntax:

```
<ausdruck1> EQ <ausdruck2>
<ausdruck1> NE <ausdruck2>
<ausdruck1> LT <ausdruck2>
<ausdruck1> LE <ausdruck2>
<ausdruck1> GT <ausdruck2>
<ausdruck1> GE <ausdruck2>
```

Die Vergleichoperatoren vergleichen <ausdruck1> mit <ausdruck2> und geben den Wert wahr (0FFFFh) zurueck, wenn die spezifizierte Bedingung erfuellt ist, oder falsch (0000h), wenn sie nicht erfuellt ist.

Operator	Rueckgabewert
EQ	wahr (0FFFFh), wenn die Ausdruecke gleich sind
NE	wahr (0FFFFh), wenn die Ausdruecke nicht gleich sind
LT	wahr (0FFFFh), wenn der linke Ausdruck kleiner als der rechte ist
LE	wahr (0FFFFh), wenn der linke Ausdruck kleiner oder gleich dem rechten ist
GT	wahr (0FFFFh), wenn der linke Ausdruck groesser als der rechte ist
GE	wahr (0FFFFh), wenn der linke Ausdruck groesser oder gleich dem rechten ist

Typisch fuer die Operatoren ist die Verwendung in bedingten Pseudooperationen und in Befehlen zur bedingten Programmsteuerung.

Beachte:

Die Operatoren EQ und NE behandeln ihre Argumente als vorzeichenbehaftete 16-Bit-Zahlen. Zahlen deren 16. Bit gesetzt ist, sind negative Zahlen (0FFFFh ist -1). Der Ausdruck -1 EQ 0FFFFh ist wahr, der Asudruck -1 NE 0FFFFh ist falsch.

Die Operatoren LT, LE, GT, GE behandeln ihre Argumente als 17-Bit-Zahlen, wobei das 17. Bit das Vorzeichen darstellt. 0FFFFh ist die groesste vorzeichenlose Zahl (65535 und nicht -1)! Der Ausdruck 1 GT -1 ist wahr (0FFFFh), der Ausdruck 1 GT 0FFFFh ist

falsch (0).

Beispiele:

```
1 EQ 0  ;falsch
1 NE 0  ;wahr
1 LT 0  ;falsch
1 LE 0  ;falsch
1 GT 0  ;wahr
1 GE 0  ;wahr
```

5.3.4.Bitoperatoren

Syntax:

```
NOT <ausdruck>
<ausdruck1> AND <ausdruck2>
<ausdruck1> OR  <ausdruck2>
<ausdruck1> XOR <ausdruck2>
```

Die logischen Operatoren fuehren bitweise logische Verknuepfungen der Ausdruecke aus. Bei Bitoperationen wird die Operation mit jedem Bit eines Ausdrucks ausgefuehrt und nicht mit dem Ausdruck als Ganzes. Der Ausdruck muss einen absoluten Wert ergeben.

Operator	Bedeutung
NOT	Invers, Negation
AND	Boolsches Und
OR	Boolsches Oder
XOR	Boolsches exklusives Oder

Beispiele:

```
NOT 11110000b                ;ist gleich 1111111100001111b
                             ;oder 00001111b
01010101b AND 11110000b      ;ist gleich 01010000b
01010101b OR  11110000b      ;ist gleich 11110101b
01010101b XOR 11110000b      ;ist gleich 10100101b
```

5.3.5._Indexoperator

Syntax:

```
[<ausdruck1>][<ausdruck2>]
```

Der Indexoperator [] addiert den Wert von <ausdruck1> zu <ausdruck2>. Dieser Operator ist identisch mit dem Operator +, mit der Ausnahme, dass <ausdruck1> wahlfrei ist.

Wenn <ausdruck1> gegeben ist, muss er links von Operator erscheinen.

<ausdruck1> kann

- **eine ganze Zahl**
- **ein absolutes Symbol**
- **ein verschieblicher Operand**

sein. Wenn <ausdruck1> nicht angegeben ist, wird der ganzzahlige Wert 0 angenommen.

Ist <ausdruck1> ein verschieblicher Operand, muss <ausdruck2>

- **eine ganze Zahl**
- **ein absolutes Symbol**

sein. Sonst kann <ausdruck2>

- **eine ganze Zahl**
- **ein absolutes Symbol**
- **ein verschieblicher Operand**

sein.

Ein typischer Anwendungsfall fuer den Indexoperator ist der indizierte Zugriff auf Elemente von Feldern, so z. B. auf ein einzelnes Zeichen in einer Zeichenkette.

Beispiele:

```
mov     al,kette[3]     ;Zugriff auf 4. Element von kette
mov     ax,feld[4]      ;Zugriff auf 5. Element von feld
mov     kette[last],al  ;Transport in das letzte Element
                        ; von kette
mov     cx,DGROUP:[1]   ;Zugriff auf 2. Byte von DGROUP
mov     cx,DGROUP:1     ;          - " -
```

Die letzten beiden Beispiele sind identisch.

5.3.6. PTR-Operator

Syntax:

<typ> **PTR** <ausdruck>

Der PTR-Operator weist der durch <ausdruck> gegebenen Marke oder Variablen den durch <typ> spezifizierten Typ zu.

<typ> kann einer der folgenden Namen oder Werte sein:

Typ	Wert	
BYTE	1	
WORD	2	
DWORD	4	nur fuer Speicheroperanden
QWORD	8	
TBYTE	10	
NEAR	0FFFFh	nur fuer Marken
FAR	0FFFEh	

<ausdruck> kann jeder Operand sein.

Der PTR-Operator wird haeufig in Vorwaertsreferenzen gebraucht. Er legt explizit fest, welche Groesse oder Entfernung diese Referenz hat. Wurde der PTR-Operator nicht verwendet, nimmt der Assembler eine Standardgroesse oder Standardentfernung an. Er ist notwendig, um Uebersetzungsfehler beim Zugriff auf Daten zu vermeiden. Zum Beispiel kann er den Zugriff auf das hoeherwertige Byte einer Variablen von Wortlaenge ermoeglichen.

Im Abschnitt 5.6. wird die Verwendung des PTR-Operators diskutiert, um die mit der strengen Typpruefung verbundenen Fehler zu vermeiden.

Solche Fehler sind:

Illegal size for item
(Illegale Groesse fuer Ausdruecke)

Operand types must match
(Operandentypen muessen angepasst werden)

Beispiele:

```
call    FAR PTR up3
mov     BYTE PTR [feld],1
add     al,BYTE PTR [wort]
```

In diesen Beispielen ueberschreibt der PTR-Operator eine vorherige Datenvereinbarung. Die Prozedur up3 koennte als NEAR erklaert worden sein, waehrend feld und wort mit der Pseudooperation DW vereinbart sein koennten.

5.3.7. Segmentpraefix

Syntax:

```
<segmentregister>:<ausdruck>
<segmentname>:<ausdruck>
<gruppenname>:<ausdruck>
```

Der Segmentpraefix (:) bewirkt das Berechnen der Adresse einer gegebenen Variablen oder Marke unter Verwendung des Beginns eines Segements oder einer Gruppe, spezifiziert durch das gegebene <segmentregister>, <segmentname> oder <gruppenname>.

<segmentname> oder <gruppenname> setzen voraus, dass dieser Name mittels SEGMENT oder GROUP definiert und unter Verwendung von ASSUME einem Segmentregister zugewiesen wurde.

<ausdruck> kann

- **ein absoluter Wert**
- **ein verschieblicher Operand**

sein.

<segmentregister> muss entweder

- **CS**
- **DS**
- **SS oder**
- **ES**

sein.

Standardmaessig wird die effektive Adresse relativ zu einem der Register DS, SS oder ES berechnet, abhaengig vom Befehl und vom Operandentyp. Alle Marken werden als NEAR angenommen. Diese Standardtypen koennen durch das Verwenden des Segmentpraefixes ueberschrieben werden.

Beispiele:

```
mov     ax,es:[bx][si]
mov     _TEXT:far_label,ax
mov     ax,DGROUP:variable
mov     al,cs:0001h
```

5.3.8. Struktur-Feldname-Operator

Syntax:

```
<variable>.<feld>
```

Der Struktur-Feldname-Operator (.) bezeichnet ein Feld innerhalb einer Struktur. <variable> ist ein Operand (meistens eine vorher

vereinbarte Strukturvariable) und <feld> der Name eines Feldes innerhalb einer Struktur. Dieser Operator ist aequivalent dem Additionsoperator (+) in Basis- oder Indexoperanden.

Beispiele:

```
inc     datum.tag
mov     zeit.min,0
mov     [bx].ziel
```

5.3.9. SHORT-Operator

Syntax:

SHORT <marke>

Dieser Operator setzt den Typ einer gegebenen Marke auf SHORT. SHORT-Marken koennen in JMP-Befehlen verwendet werden, wenn die Entfernung von der Marke zu dem Befehl nicht groesser als 127 Bytes ist. Befehle, die SHORT-Marken benutzen, sind 1 Byte kuerzer als die gleichen Befehle, die NEAR-Marken verwenden.

Beispiel:

```
jmp     SHORT zyklus          ;Sprung < 127 Bytes
```

5.3.10. THIS-Operator

Syntax:

THIS <typ>

Dieser Operator erzeugt einen Operanden, dessen Offset und Segmentwert gleich dem aktuellen Speicherplatzzaehlerwert und dessen Typ der angegebene ist.

<typ> kann einer der folgenden sein:

- **BYTE**
- **WORD**
- **DWORD**
- **QWORD**
- **TBYTE**
- **NEAR**
- **FAR**

Dieser Operator wird haeufig in EQU- oder =-Pseudooperationen verwendet, um Marken und Variable zu erzeugen. Der Operator THIS ist der Pseudooperation LABEL aehnlich.

Beispiele:

```
1.    tag     EQU     THIS BYTE
```

Dieses Beispiel ist aequivalent der Anweisung

```
      tag     LABEL   BYTE
```

```
2.    test    =       THIS NEAR
```

Das gleiche Ergebnis wuerde mit

```
      test    LABEL   NEAR
```

erzielt werden.

5.3.11. HIGH- und LOW-Operator

Syntax:

```
HIGH <ausdruck>
LOW <ausdruck>
```

Der Operator HIGH gibt die hoeherwertigen und LOW die niederwertigen 8 Bits von <ausdruck> zurueck. <ausdruck> kann jeder Wert sein.

Beispiele:

```
      mov     ah,HIGH wort_wert     ;hoeherwertiges Byte von
                                    ; wort_wert -> ah
      mov     al,LOW 0ABCDh         ; 0CDh -> al
```

5.3.12. SEG-Operator

Syntax:

```
SEG <ausdruck>
```

Der SEG-Operator gibt den Segmentwert von <ausdruck> zurueck. <ausdruck> kann

- eine Marke
- eine Variable
- ein Segmentname
- ein Gruppenname
- ein anderes Symbol

sein.

Beispiele:

```
mov     ax,SEG variable
mov     ax,SEG marke
```

5.3.13. OFFSET-Operator

Syntax:

OFFSET <ausdruck>

Das Ergebnis dieses Operators ist der Offset von <ausdruck>. <ausdruck> kann

- **eine Marke**
- **eine Variable**
- **ein Segmentname**
- **ein anderes Symbol**

sein. Der zurueckgegebene Wert ist die Anzahl Bytes zwischen diesem Punkt und dem Anfang des Segments, in dem dieser definiert ist. Ist ein Segmentname angegeben worden, wird die aktuelle Groesse dieses Segments zurueckgegeben.

Es besteht auch die Moeglichkeit, den Offset eines Punktes in einem anderen als dem aktuellen Segment bzw. der aktuellen Gruppe mittels Segmentpraefix zu erhalten.

Beispiele:

```
mov     bx,OFFSET up3
mov     bx,OFFSET dgroup:feld
```

Der Rueckgabewert ist immer ein relativer Wert, der der Aenderung durch LINK unterworfen ist, wenn das Programm gebunden wird.

5.3.14. TYPE-Operator

Syntax:

TYPE <ausdruck>

Das Ergebnis dieses Operators ist eine Zahl, die den Typ von <ausdruck> darstellt. Wenn der Ausdruck eine Variable ist, gibt der Operator die Groesse des Operanden in Bytes zurueck. Ist der Ausdruck eine Marke, wird fuer NEAR der Wert 0FFFFh und fuer FAR 0FFFEh zurueckgegeben.

Dieser Rueckgabewert kann fuer den PTR-Operator zum spezifizieren des Typs benutzt werden (siehe 2. Beispiel).

Beispiele:

```
mov     ax,TYPE feld
jmp     (TYPE m1) PTR m2
```

5.3.15. .TYPE-Operator

Syntax:

```
.TYPE <ausdruck>
```

Der Operator .TYPE gibt ein Byte zurueck, das den Modus und die Reichweite von <ausdruck> definiert. <ausdruck> ist ungueltig, wenn .TYPE eine Null zurueck gibt.

In der folgenden Tabelle sind die Variablenattribute aufgezeigt, die in den Bitpositionen 0, 1, 5 und 7 verschluesselt sind:

Bit-position	Bit = 0	Bit = 1
0	nicht programmrelativ	programmrelativ
1	nicht datenrelativ	datenrelativ
5	nicht definiert	definiert
7	Lokal oder PUBLIC	EXTRN

Sind beide, das Reichweitebit (7) und das Definitionsbit (5) Null, dann ist der Ausdruck ungueltig.

Der .TYPE-Operator wird haeufig in bedingten Pseudooperationen verwendet, um ein Argument zu testen und entsprechende Entscheidungen ueber den Programmlauf zu treffen.

Beispiel:

```
x       DB      12
z       EQU     .TYPE x
```

In diesem Beispiel wird z auf den Wert 22h (00100010b) gesetzt. Bit 0 ist nicht gesetzt, da x nicht programmrelativ ist, Bit 1 ist gesetzt, weil x datenrelativ ist. Bit 5 ist gleich 1, weil x definiert ist und Bit 7 muss 0 sein, da x eine lokale Variable ist. Die uebrigen Bits sind nicht gesetzt.

5.3.16. LENGTH-Operator

Syntax:

LENGTH <variable>

Der Operator LENGTH liefert als Ergebnis die Anzahl BYTE-, WORD-, DWORD-, QWORD- oder TBYTE-Elemente in <variable>. Die Groesse jedes Elementes haengt vom definierten Typ der Variablen ab. Nur fuer Variable, die unter Verwendung des DUP-Operators definiert wurden, ist der Wert groesser als 1. Der Wert ist immer die Zahl, die dem **ersten DUP-Operator** vorausgeht.

Fuer die folgenden Beispiele werden diese Definitionen angenommen:

```
feld    DW      100 DUP(1)
tabelle DW      100 DUP(1,10 DUP(?))
```

Beispiele:

```
mov     cx,LENGTH feld
mov     cx,LENGTH tabelle
```

In beiden Beispielen wird der Wert 100 in das Register cx geladen. Im zweiten Beispiel haengt der Rueckgabewert nicht von den geschachtelten DUP-Anweisungen ab!

5.3.17. SIZE-Operator

Syntax:

SIZE <variable>

Das Ergebnis dieses Operators ist die totale Anzahl Bytes, die dieser Variablen zugewiesen wurden. Das Ergebnis ergibt sich aus dem Wert von LENGTH mal dem Wert von TYPE.

Im Beispiel wird folgende Defintion angenommen:

```
feld    DW 100 DUP(1)
```

Beispiel:

```
mov     bx,SIZE feld
```

Es wird der Wert 200 nach bx gebracht.

5.3.18. WIDTH-Operator

Syntax:

WIDTH {<recordfeldname>|<record>}

Der Operator WIDTH gibt die Groesse in Bits eines Recordfeldes oder eines Record an. <recordfeldname> muss der Name eines in einem Record definierten Feldes sein.

Fuer die naechsten Beispiele werden die folgende Recordvereinbarung und -definition angenommen:

```
rtyp    RECORD  feld1:3,feld2:6,feld3:7
rec1    rtyp <>
```

Beispiele:

```
wid1    =       WIDTH feld1     ; = 3
wid2    =       WIDTH feld2     ; = 6
wid3    =       WIDTH feld3     ; = 7
widrec  =       WIDTH rtyp      ; = 16
```

Zur Erinnerung: Der Feldname repraesentiert die Bitposition des Feldes im Record, z. B. feld1 ist gleich 13 (die Groesse von feld2 plus die Groesse von feld3). Aber WIDTH feld1 ist 3.

5.3.19. MASK-Operator

Syntax:

```
MASK {<recordfeldname>|<record>}
```

Das Ergebnis des Operators MASK ist eine Bitmaske. Ein Bit in der Maske enthaelt eine 1, wenn das Bit einem Feldbit entspricht. Alle anderen Bits sind 0.

<recordfeldname> muss der Name eines Feldes, definiert in einem Record, sein.

Im naechsten Beispiel wird die folgende Recordvereinbarung und -definition angenommen:

```
rtyp    RECORD  feld1:3,feld2:6,feld3:7
rec1    rtyp    <>
```

Beispiele:

```
m1      =       MASK feld1      ;E000h  (1110000000000000b)
m2      =       MASK feld2      ;1F80h  (1111110000000b)
m3      =       MASK feld3      ;007Fh  (1111111b)
mrec    =       MASK rtyp       ;0FFFFh (1111111111111111b)
```

5.4. Bestimmung von Ausdruecken und Rangfolge der Operatoren

Ausdruecke werden in Uebereinstimmung mit den Regeln ueber Vorrang und Reihenfolge der Operatoren ausgewertet. Operationen mit

einer hoeheren Prioritaet werden zuerst ausgefuehrt. Operationen mit gleicher Prioritaet werden von links nach rechts abgearbeitet. Die Standardreihenfolge kann durch Klammern veraendert werden. In Klammern eingeschlossene Operationen werden immer vor den benachbarten Operationen ausgefuehrt.

In der folgenden Tabelle werden alle Operatoren in ihrer Rangfolge aufgelistet. Operatoren auf der selben Zeile haben die gleiche Prioritaet.

Prioritaet	Operatoren
(Hoechste) 1	LENGTH, SIZE, WIDTH, MASK, (), [], <>
2	. (Struktur-Feldname-Operator)
3	: (Segmentpraefix)
4	PTR, OFFSET, SEG, TYPE, THIS
5	HIGH, LOW
6	+, - (Vorzeichen)
7	*, /, MOD, SHL, SHR
8	+, -
9	EQ, NE, LT, LE, GT, GE
10	NOT
11	AND
12	OR, XOR
13 (Niedrigste)	SHORT, .TYPE

Beispiele:

```
8/4*2                   ;= 4
8/(4*2)                 ;= 1
8+4*2                   ;=16
(8+4)*2                 ;=24
8 EQ 4 AND 2 LT 3       ;0000h (falsch)
8 EQ 4 OR  2 LT 3       ;0FFFFh (wahr)
```

5.5. Vorwaertsreferenzen

Obwohl der Assembler Vorwaertsreferenzen auf Marken, Variable, Segmentnamen und andere Symbole erlaubt, koennen sie zu Assemblerfehlern fuehren, wenn sie nicht korrekt verwendet werden.

Eine Vorwaertsreferenz ist jede Verwendung eines Namens, bevor dieser definiert wurde.

Zum Beispiel waere die Märke weiter in dem JMP-Befehl eine Vorwaertsreferenz:

```
            jmp     weiter
            mov     ax,0
             .
             .
             .
weiter:
```

Trifft der Assembler im Pass 1 auf einen undefinierten Namen, nimmt er diesen als eine Vorwaertsreferenz an. Ist nur der Name vorhanden, trifft der Assembler Annahmen ueber Typ und Segmentregister, um Kode oder Daten fuer diese Anweisung zu generieren.

In oben stehendem Beispiel nimmt MASM an, dass weiter eine Marke vom Typ NEAR ist. Er generiert 3 Bytes fuer den Befehlskode.

Der Assembler trifft seine Annahmen entsprechend der Anweisung, die die Vorwaertsreferenz enthaelt. Sind diese Annahmen nicht korrekt, koennen Fehler auftreten. Waere zum Beispiel weiter eine FAR-Marke und keine NEAR-Marke, verursacht die im Pass 1 getroffene Annahme einen Phasenfehler. Denn im Pass 2 wuerden 5 Bytes fuer den Befehlskode dieser JMP-Anweisung generiert, im Pass 1 aber nur 3 Bytes.

Um Fehler mit Vorwaertsreferenzen zu vermeiden, sollten die Operatoren **:**, **PTR** und **SHORT** verwendet werden, wenn die Notwendigkeit besteht, die vom Assembler getroffenen Annahmen zu ueberschreiben. Im folgenden sind die Anwendungsfaelle fuer diese Operatoren aufgelistet.

- Erfolgt eine Vorwaertsreferenz auf eine Variable, relativ zu den Registern ES, SS oder CS, dann sollte der Segmentpraefix zum Spezifizieren des Segmentregisters, des Segments oder der Gruppe der Variablen verwendet werden.

  Beispiele:

  ```
  mov   ax,ss:stackoben
  inc   data:zeit[1]
  add   ax,dgroup:_I
  ```

  Wuerde hier der Segmentpraefix nicht angegeben, nimmt der Assembler die Variable relativ zum DS-Register an.

- Stellt die Vorwaertsreferenz eine Befehlsmarke in einem

JMP-Befehl dar, dann verwendet man den Operator SHORT, wenn der Befehl weniger als 128 Bytes von der Referenz entfernt ist.

Beispiel:

```
jmp   SHORT weiter
```

Wuerde SHORT nicht verwendet, nimmt der Assembler eine Entfernung groesser als 128 Bytes an. Das wuerde keinen Fehler verursachen. Aber der Assembler generiert einen unnoetigen NOP-Befehl.

- Stellt die Vorwaertsreferenz eine Befehlsmarke in einem CALL- oder JMP-Befehl dar, dann sollte der PTR-Operator benutzt werden, um den Typ der Marke festzulegen.

Beispiele:

```
call  FAR PTR druck
jmp   FAR PTR ende
```

Der Assembler nimmt standardmaessig an, dass die Marke den Typ NEAR hat. Folglich muss der PTR-Operator fuer NEAR-Marken nicht verwendet werden. Hat eine Marke den Typ FAR, muss FAR PTR geschrieben werden. Sonst wird ein Phasenfehler erzeugt.

- Ist eine Vorwaertsreferenz ein Segmentname mit einem Segmentpraefix, sollte die GROUP-Anweisung verwendet werden, um den Segmentnamen mit einem Gruppennamen zu verbinden. Danach ist diesem Gruppennamen mit ASSUME ein Segmentregister zuzuweisen.

Beispiel:

```
dgroup  GROUP   stack
        ASSUME  ss:dgroup

code    SEGMENT
          .
          .
          .
        mov     ax,stack
          .
          .
          .
```

Wird die Gruppe nicht mit dem Segmentnamen verbunden, ignoriert der Assembler den Segmentpraefix und benutzt das Standardsegmentregister fuer die Variable. Das ergibt meistens einen Phasenfehler im Pass 2.

5.6. Strenge Typpruefung fuer Speicheroperanden

Der Assembler fuehrt eine strenge Syntaxpruefung fuer alle Befehlsanweisungen durch, einschliesslich der strengen Typpruefung fuer Operanden, die auf einen Speicherplatz Bezug nehmen.

Folglich muss jeder verschiebliche Operand in einem Befehl, der auf einen impliziten Datentyp wirkt, entweder genau diesen Datentyp oder eine explizite Datentypueberschreibung (PTR-Operator) besitzen.

Im folgenden Beispiel wurde die Variable kette fehlerhaft in einem MOV-Befehl verwendet:

```
kette   DB      "Eine Meldung"
        .
        .
        .
        mov     ax,kette[1]
```

Im Ergebnis wird die Fehlermeldung

```
Operand types must match
(Operandentypen muessen angepasst werden)
```

generiert, weil kette den Typ BYTE hat, der Befehl erwartet aber eine Variable vom Typ WORD. Dieser Fehler wird vermieden, wenn der PTR-Operator zum Ueberschreiben des Variablentyps verwendet wird. Der folgende Befehl wird korrekt uebersetzt:

```
        mov     ax,WORD PTR kette[1]
```

6. Vereinbaren von Globalen

6.1. Einleitung

Die Pseudooperationen zur Vereinbarung von Globalen erlauben es, Marken, Variable und absolute Symbole zu definieren, auf die global zugegriffen werden kann, d. h. sie koennen von allen Moduln in einem Programm angesprochen werden.

Die beiden Pseudooperationen zur Vereinbarung von Globalen sind PUBLIC und EXTRN.

Die Pseudooperation PUBLIC wird verwendet zur Vereinbarung von Eintrittspunkten und zur Umwandlung von lokal definierten Symbolen in globale Symbole, um sie in anderen Moduln verfuegbar zu machen.

Die Pseudooperation EXTRN dient der Vereinbarung einer externen Bezugnahme, d. h. der Name eines globalen Symbols und sein Typ werden in einer Quelldatei verfuegbar gemacht, so dass sie in dieser Datei verwendet werden koennen.

Jedes globale Symbol muss genau eine PUBLIC-Vereinbarung in

einer Quelldatei des Programms haben. Ein globales Symbol kann in beliebig vielen anderen Quelldateien EXTRN-Vereinbarungen haben.

6.2. PUBLIC

Syntax:

PUBLIC <name>[,<name>,...]

Durch die Pseudooperation PUBLIC werden die Variablen, Marken oder absoluten Symbole, die durch <name> spezifiziert wurden, in allen anderen Moduln des Programms verfuegbar gemacht.

<name> muss der Name einer Variablen, einer Marke oder eines absoluten Symbols sein, definiert innerhalb der aktuellen Quelldatei. Absolute Symbole koennen nur ganze Zahlen oder Zeichenkettenwerte der Laenge 1 oder 2 Bytes sein.

Der Assembler wandelt alle kleinen Buchstaben in grosse, bevor der Name in die Objektdatei uebernommen wird. Die Schalter /ML und /MX koennen in der Befehlszeile von MASM angegeben werden, um die Schreibweise der PUBLIC- und EXTRN-Symbole bei der Uebernahme in die Objektdatei beizubehalten. Siehe dazu auch die Abschnitte 10.3.7. und 10.3.8., in denen diese beiden Schalter beschrieben sind.

Symbole muessen als PUBLIC vereinbart werden, bevor sie fuer den symbolischen Test verwendet werden koennen. In der Beschreibung der Testhilfe SYMDEB ist detailiert beschrieben, wie die Symboldateien mit diesem Programm vorbereitet und verwendet werden.

Beispiel:

```
        PUBLIC  wahr,status,beginn,loe
wahr    =       0FFFFh
status  DB      1
beginn  LABEL   FAR
loe     PROC    NEAR
```

In diesem Beispiel wird ein absolutes Symbol, eine Variable, eine Marke und eine Prozedur als PUBLIC vereinbart.

6.3. EXTRN

Syntax:

EXTRN <name>:<typ>[,<name>:<typ>,...]

Die Pseudooperation EXTRN definiert eine externe Bezugnahme mit dem spezifizierten Namen und Typ.

Eine externe Bezugnahme ist eine Variable, Marke oder Symbol,

das mit PUBLIC in einem anderen Modul des Programms vereinbart wurde. <typ> muss gleich dem in der entsprechenden Definition angegebenen Typ sein. Er kann einer der folgenden sein:

- BYTE
- WORD
- DWORD
- QWORD
- TBYTE
- NEAR
- FAR
- ABS

Der Typ ABS ist fuer Symbole, die eine absolute Zahl darstellen.

Die aktuelle Adresse ist bis zum Binden der Objektdatei unbestimmt. Der Assembler trifft deshalb eine Annahme ueber ein Standardsegment fuer den externen Verweis. Diese Annahme ist abhaengig davon, wo die Anweisung EXTRN im Modul steht.

Steht EXTRN innerhalb eines Segments, dann wird der externe Punkt als relativ zu diesem Segment angenommen. Die zugehoerige PUBLIC-Anweisung (in einem anderen Modul) muss sich in einem Segment mit demselben Namen und denselben Attributen befinden.

Steht EXTRN ausserhalb aller Segmente, wird keine Annahme darueber getroffen, zu welchem Segment dieser Punkt relativ ist. Die zugehoerige PUBLIC-Anweisung kann sich in jedem Segment in jedem Modul befinden.

In jedem Fall kann der Segmentpraefix zum Ueberschreiben des Standardsegments fuer die Variable oder Marke benutzt werden.

Beispiel:

```
EXTRN   tag:near
EXTRN   mon:word,dat:dword
```

6.4. Programmbeispiel

Die folgende Quelldatei zeigt ein Programm, das PUBLIC- und EXTRN-Vereinbarungen zum Zugriff auf Befehlsmarken verwendet. Das Programm besteht aus zwei Moduln, genannt prog und up. Der Modul prog ist der Programminitialisierungsteil. Die Ausfuehrung beginnt an der Befehlsmarke beginn in prog und geht zur Befehlsmarke druck in up ueber. In up wird ein Systemruf verwendet, um Hallo auf den Bildschirm zu bringen. Die Abarbeitung endet mit dem Befehl exit im Modul prog.

Beispiel:

Hauptprogramm_prog:

```
                NAME    prog
                PUBLIC  exit
                EXTRN   druck:near

        stack   SEGMENT word stack 'STACK'
                DW      64 DUP(?)
        stack   ENDS

        data    SEGMENT word public 'DATA'
        data    ENDS

        code    SEGMENT byte public 'CODE'
                ASSUME  cs:code, ds:data
        beginn:
                mov     ax,data         ;Laden Segmentadresse
                mov     ds,ax           ; --> DS
                jmp     druck           ;Sprung zu druck im
                                        ;anderen Modul
        exit:
                mov     ah,4Ch          ;Aufruf Programmende
                int     21h
        code    ENDS
                END
```

Unterprogramm_up:

```
                NAME    up
                PUBLIC  druck
                EXTRN   exit:near

        data    SEGMENT word public 'DATA'
        meld    DB      "Hallo",13,10,"$"
        data    ENDS

        code    SEGMENT byte public 'DATA'
                ASSUME  cs:code,ds:data
        druck:
                mov     dx,OFFSET meld  ;Laden Adresse Anzeige-
                                        ;text
                mov     ah,09h          ;Zeichenkettenanzeige-
                                        ; funktion
                int     21h
                jmp     exit            ;zurueck in anderen
                                        ;Modul
        code    ENDS
                END
```

In diesem Beispiel wird das Symbol exit als PUBLIC im Modul prog vereinbart, so dass darauf in anderen Quellmoduln (im Beispiel up) zugegriffen werden kann. Der Modul prog enthaelt auch eine EXTRN-Vereinbarung fuer das Symbol druck. Diese Vereinbarung definiert druck als eine NEAR-Marke. Sie kann unter der Voraus-

setzung, dass sie in einem anderen Quellmodul angelegt und als PUBLIC vereinbart wurde, in prog verwendet werden. Ein jmp-Befehl hat diese Marke als Operand.

Das Symbol druck wird im Modul up als Public vereinbart. Damit kann diese Marke in anderen Moduln angesprochen werden. exit ist als NEAR-Marke definiert. Es wird angenommen, dass sie in einem anderen Modul angelegt und als PUBLIC vereinbart ist.

Bevor dieses Programm abgearbeitet werden kann, muessen die Quelldateien einzeln uebersetzt und dann mittels LINK gebunden werden.

7. Bedingte Pseudooperationen

7.1. Einleitung

Der Makroassembler bietet zwei Typen von bedingten Pseudooperationen an. Die Pseudooperationen fuer die bedingte Uebersetzung testen eine spezifizierte Bedingung und uebersetzen einen Anweisungsblock, wenn die Bedingung wahr ist. Bedingte Fehlerpseudooperationen testen eine spezifizierte Bedingung und generieren einen Fehler, wenn die Bedingung wahr ist.

Beide Arten von bedingten Pseudooperationen testen die Bedingung nur zur Uebersetzungszeit. Sie koennen die Bedingungen nicht zur Abarbeitungszeit testen, weil sie zu dieser Zeit nicht bekannt sind. Es koennen nur solche Ausdruecke verglichen und getestet werden, die waehrend der Uebersetzung als Ergebnis eine Konstante ergeben.

Weil Makros und bedingte Pseudooperationen oft gemeinsam verwendet werden, ist es notwendig, auf das Kapitel 8 zu verweisen, um einige der Beispiele in diesem Kapitel zu verstehen. Bedingte Pseudooperationen werden haeufig mit den speziellen Makrooperatoren verwendet. Sie sind im Abschnitt 8.3. beschrieben.

7.2. Pseudooperationen fuer die bedingte Uebersetzung

Es gibt folgende Pseudooperationen fuer die bedingte Uebersetzung:

- IF
- IFE
- IF1
- IF2
- IFDEF
- IFNDEF
- IFB
- IFNB
- IFIDN
- IFDIF
- ELSE
- ENDIF

Die Pseudooperationen IF, ENDIF und ELSE koennen zum Einschliessen von Anweisungen, die fuer die bedingte Uebersetzung in Betracht kommen, verwendet werden. Der bedingte Block hat folgende Form:

```
IF
        <anweisungen>
[ ELSE
        <anweisungen>]
ENDIF
```

Die dem IF folgenden <anweisungen> koennen alle gueltigen Anweisungen, einschliesslich andere bedingte Bloecke sein. Die Pseudooperation ELSE und die zugehoerigen Anweisungen sind wahlfrei. ENDIF beendet den Block.

Die Anweisungen im bedingten Block werden nur dann uebersetzt, wenn die bei dem zugehoerigen IF spezifizierte Bedingung erfuellt ist. Enthaelt der bedingte Block eine ELSE-Anweisung, dann werden nur die Anweisungen bis zum ELSE uebersetzt. Die dem ELSE folgenden Anweisungen werden bei nichterfuellter Bedingung uebersetzt. Jeder bedingte Block muss mit einer ENDIF-Anweisung abgeschlossen werden. Zu jedem IF ist hoechstens eine ELSE-Anweisung erlaubt.

IF-Anweisungen koennen bis zu 255 mal geschachtelt werden. **Eine ELSE-Anweisung wird immer der naechsten vorhergehenden IF-Anweisung zugeordnet, die noch kein ELSE hat.**

7.2.1. IF und IFE

Syntax:

```
IF <ausdruck>
IFE <ausdruck>
```

Die Pseudooperationen IF und IFE testen den Wert eines Ausdruckes. Die IF-Anweisung erlaubt die Uebersetzung, wenn der Wert des Ausdruckes wahr (nicht Null) ist. Die IFE-Anweisung erlaubt die Uebersetzung, wenn der Wert des Ausdruckes falsch (Null) ist. Der Ausdruck muss einen absoluten Wert ergeben und darf keine Vorwaertsreferenzen enthalten.

Beispiel:

```
IF      test
        EXTRN   dump:FAR
        EXTRN   trace:FAR
        EXTRN   breakpoint:FAR
ENDIF
```

In diesem Beispiel werden die Variablen innerhalb des Blockes nur dann als EXTRN vereinbart, wenn das Symbol test den Wert

wahr (nicht Null) ergibt.

7.2.2. IF1 und IF2

Syntax:

```
IF1
IF2
```

Die Pseudooperationen IF1 und IF2 testen den aktuellen Assemblerpass. Die IF1-Anweisung erlaubt die Uebersetzung nur im Pass 1, die IF2-Anweisung nur im Pass 2. Beide Pseudooperationen haben keine Argumente.

Beispiel:

```
IF1
        %OUT Beginn des Pass 1
ELSE
        %OUT Beginn des Pass 2
ENDIF
```

7.2.3. IFDEF und IFNDEF

Syntax:

```
IFDEF <name>
IFNDEF <name>
```

Die Pseudooperationen IFDEF und IFNDEF testen, ob der gegebene <name> definiert worden ist oder nicht. IFDEF erlaubt die Uebersetzung nur, wenn <name> eine Marke, eine Variable oder ein Symbol ist. IFNDEF erlaubt die Uebersetzung, wenn <name> nicht definiert wurde.

<name> kann jeder gueltige Name sein. Es ist zu beachten, ein Name, der eine Vorwaertsreferenz darstellt, ist im Pass 1 undefiniert und im Pass 2 definiert.

Beispiel:

```
IFDEF   puffer
        puffer1 DB      10 DUP(?)
ENDIF
```

In diesem Beispiel wird puffer1 nur dann zugewiesen, wenn vorher puffer definiert worden ist. Im folgenden wird eine Moeglichkeit fuer die Verwendung dieses bedingten Blockes gezeigt. puffer wird in der Quelldatei nicht definiert, sondern es wird bei Bedarf beim Aufruf von MASM der Schalter /D<symbol> genutzt. Ist dieser Block beispielsweise in der Quelldatei test.asm enthalten, koennte die Befehlszeile des Assemblers

```
MASM test /Dpuffer;
```

heissen. Das Symbol puffer waere damit definiert. Als Ergebnis wuerde durch den bedingten Block der Bereich puffer1 angelegt. Wird puffer1 nicht benoetigt, dann lautet die Befehlszeile

```
MASM test;
```

7.2.4. IFB und IFNB

Syntax:

```
IFB <argument>
IFNB <argument>
```

Durch die Pseudooperationen IFB und IFNB wird <argument> getestet. IFB erlaubt die Uebersetzung bei leerem Argument und IFNB bei nichtleerem Argument.

<argument> kann

- ein Name
- eine Zahl
- ein Ausdruck

sein. **Die spitzen Klammern (<>) muessen geschrieben werden.**

Diese Pseudooperationen werden in Makrodefinitionen verwendet. Sie steuern die bedingte Assemblierung der Anweisungen in Abhaengigkeit davon, ob Parameter im Makroaufruf uebergeben worden sind. In solchen Faellen sollte <argument> einer der in der Pseudooperation MACRO aufgelisteten symbolischen Parameter sein.

Beispiel:

```
pushall MACRO   reg1,reg2,reg3,reg4,reg5,reg6
        IFNB    <reg1>              ;;wenn Parameter nicht
                                    ;;leer
                push    reg1        ;;push 1 Register
                pushall reg2,reg3,reg4,reg5,reg6
                                    ;;und wiederholen
        ENDIF
        ENDM

pushall ax,bx,si,ds
pushall cs,es
```

pushall ist ein rekursiver Makro. Er enthaelt den Aufruf von sich selbst solange, bis ein leeres Argument auftritt. Ein Register oder eine Liste von Registern (bestehend aus bis zu sechs Registern) kann an den Makro uebergeben werden.

7.2.5. IFIDN und IFDIF

Syntax:

```
IFIDN <argument1>,<argument2>
IFDIF <argument1>,<argument2>
```

Die Pseudooperationen IFIDN und IFDIF vergleichen <argument1> und <argument2>. IFIDN erlaubt die Uebersetzung bei identischen und IFDIF bei verschiedenen Argumenten.

Identisch heisst, jedes Zeichen von <argument1> ist gleich dem zugehoerigen Zeichen in <argument2>. **Die spitzen Klammern (<>) muessen geschrieben werden.** Die Argumente sind durch ein Komma (,) voneinander zu trennen.

Diese Pseudooperationen werden in Makrodefinitionen verwendet. Sie steuern die bedingte Assemblierung der Anweisungen in Abhaengigkeit davon, ob Parameter im Makroaufruf uebergeben worden sind. In solchen Faellen sollten die Argumente die in der Pseudooperation MACRO aufgelisteten symbolischen Parameter sein.

Beispiel:

```
divis   MACRO   zaehler,nenner
        IFDIF   <nenner>,<0>    ;;wenn nicht Division : 0
                mov     ax,zaehler      ;;Division ax
                mov     bx,nenner       ;;durch bx
                div     bx              ;;Ergebnis in
                                        ;; Akkumulator
        ENDIF
        ENDM

divis   6,%test
```

In diesem Makro wird die Pseudooperation IFDIF benutzt, um eine Division durch 0 zu verhindern. Beim Makroaufruf wird der zweite Parameter mit einem Prozentzeichen (%) angegeben, so dass der Wert des Parameters und nicht sein Name ausgewertet wird. Im Abschnitt 8.3.4. wird dieser Operator erklaert.

Wurde der Parameter test vorher mit der Anweisung

```
test    EQU     0
```

definiert, dann ist die Bedingung erfuellt. Der Kode in diesem Block wird nicht assembliert. Lautet die Definitionsanweisung

```
test    DW      0
```

wird der Fehler 42

```
Constant was expected
(Eine Konstante wird erwartet)
```

generiert, weil der Assembler keine Moeglichkeit hat, den Laufzeitwert der Variablen test zu ermitteln. Bedingte Pseudooperationen koennen nur die zum Uebersetzungszeitpunkt bekannten Konstanten auswerten.

7.3. Bedingte Fehlerpseudooperationen

Bedingte Fehlerpseudooperationen werden zum Testen von Programmen und Pruefen von Uebersetzungsfehlern benutzt. Durch Einfuegen einer bedingten Fehlerpseudooperation an einer Schluesselstelle im Quellkode, werden zum Assemblerzeitpunkt an dieser Stelle Bedingungen getestet (z. B. Grenzbedingungen in Makros).

Im folgenden sind die bedingten Fehlerpseudooperationen und die von ihnen erzeugten Fehler aufgelistet:

Pseudo-operation	Nummer und Meldung
.ERR1	87 Forced error - pass1 (Erzwungener Fehler - Pass 1)
.ERR2	88 Forced error - pass2 (Erzwungener Fehler - Pass 2)
.ERR	89 Forced error (Erzwungener Fehler)
.ERRE	90 Forced error - expression equals 0 (Erzwungener Fehler - Ausdruck = 0)
.ERRNZ	91 Forced error - expression not equal 0 (Erzwungener Fehler - Ausdruck ≠ 0)
.ERRNDEF	92 Forced error - symbol not defined (Erzwungener Fehler - Symbol nicht definiert)
.ERRDEF	93 Forced error - symbol defined (Erzwungener Fehler - Symbol definiert)
.ERRB	94 Forced error - string blank (Erzwungener Fehler - Zeichenkette leer)
.ERRNB	95 Forced error - string not blank (Erzwungener Fehler - Zeichenkette nicht leer)
.ERRIDN	96 Forced error - strings identical (Erzwungener Fehler - Zeichenketten identisch)
.ERRDIF	97 Forced error - strings different (Erzwungener Fehler - Zeichenketten verschieden)

Diese durch bedingte Fehlerpseudooperationen generierten schweren Fehler veranlassen den Assembler den Rueckkehrcode 7 zurueckzugeben und den Objektmodul zu loeschen. Alle bedingten Fehlerpseudooperatien ausser .ERR1 generieren schwere Fehler.

7.3.1 .ERR, .ERR1 und .ERR2

Syntax:

```
.ERR
.ERR1
.ERR2
```

Die Pseudooperationen .ERR, .ERR1 und .ERR2 erzeugen an der Stelle ihres Auftretens in der Quelldatei einen Fehler. Die .ERR-Anweisung generiert unabhaengig vom Pass .ERR1 und .ERR2 nur in ihrem zugehoerigen Pass einen Fehler.

.ERR1 erscheint nur auf dem Bildschirm oder in der Listendatei, wenn mit dem /D-Schalter eine Pass 1 - Liste angefordert wurde. Im Gegensatz zu allen anderen Fehlerpseudooperationen erzeugt diese keinen schwereren Fehler.

Diese Anweisungen koennen innerhalb von bedingten Bloecken verwendet werden, um zu sehen, welche Bloecke erweitert worden sind.

Beispiel:

```
IFDEF   dcp
          .
          .
          .
ELSE
        IFDEF   mutos
                  .
                  .
                  .
        ELSE
                .ERR
        ENDIF
ENDIF
```

Dieses Beispiel sichert, dass entweder das Symbol dcp oder das Symbol mutos definiert ist. War keines von beiden definiert, wird die innere ELSE-Bedingung uebersetzt und damit eine Fehlermeldung generiert. Da die .ERR-Anweisung benutzt wurde, wird in jedem Pass ein Fehler generiert. Wuerde .ERR2 benutzt, erschiene nur ein schwerer Fehler. Bei der Verwendung von .ERR1 wird nur eine Warnung gegeben.

7.3.2. .ERRE und .ERRNZ

Syntax:

```
.ERRE <ausdruck>
.ERRNZ <ausdruck>
```

Die Pseudooperationen .ERRE und .ERRNZ testen den Wert eines Ausdrucks. .ERRE generiert einen Fehler, wenn der Ausdruck falsch (Null) ist. .ERRNZ erzeugt einen Fehler, wenn der Ausdruck wahr (nicht Null) ist. <ausdruck> muss einen absoluten Wert ergeben und darf keine Vorwaertsreferenz enthalten.

Beispiel:

```
puffer  MACRO   anzahl,pname
        .ERRE   anzahl LE 128           ;;Zuweisen Speicher
        pname   DB      anzahl DUP(0)   ;;aber nicht mehr
        ENDM                            ;;als 128 Bytes

puffer  128,puffer1     ;Datenzuweisung ohne Fehler
puffer  129,puffer2     ;es wird ein Fehler erzeugt
```

In diesem Beispiel wird die Pseudooperation .ERRE verwendet, um die Grenzen eines an den Makro puffer uebergebenen Parameters zu pruefen. Ist anzahl kleiner oder gleich 128, wird der durch die Fehlerpseudooperation getestete Ausdruck wahr (nicht Null). Es wird kein Fehler generiert. Ist anzahl groesser als 128, wird der Ausdruck falsch (Null), es wird ein Fehler erzeugt.

7.3.3. .ERRDEF und .ERRNDEF

Syntax:

```
.ERRDEF <name>
.ERRNDEF <name>
```

Die Pseudooperationen .ERRDEF und .ERRNDEF testen, ob <name> definiert worden ist oder nicht. Durch .ERRDEF wird ein Fehler erzeugt, wenn <name> als Marke, Variable oder Symbol definiert wurde. Wurde <name> bis zu dieser Stelle noch nicht definiert, generiert .ERRNDEF einen Fehler.

<name> als Vorwaertsreferenz wird im 1. Pass als undefiniert und im 2. Pass als definiert betrachtet.

Beispiel:

```
.ERRDEF symbol
IFDEF   config1
          .
          .     symbol  EQU     0
          .
ENDIF
IFDEF   config2
          .
          .     symbol  EQU     1
          .
ENDIF
.ERRNDEF symbol
```

In diesem Beispiel sorgt die Pseudooperation .ERRDEF am Anfang des bedingten Blockes dafuer, dass symbol vor dem Eintreten in den Block nicht definiert worden ist. Am Ende stellt .ERRNDEF sicher, dass symbol irgendwo innerhalb des Blockes definiert wurde.

2.3.4 .ERRB und .ERRNB

Syntax:

```
.ERRB <zeichenkette>
.ERRNB <zeichenkette>
```

Die Pseudooperationen .ERRB und .ERRNB testen die gegebene Zeichenkette. .ERRB erzeugt bei leerer und .ERRNB bei nicht leerer Zeichenkette einen Fehler.

<zeichenkette> kann

- ein Name
- eine Zahl
- ein Ausdruck

sein. Die spitzen Klammern (<>) muessen geschrieben werden.

Beispiel:

```
beisp   MACRO   param,test
        .ERRB   <param>         ;;Fehler, wenn kein
                                ;;Parameter uebergeben
        .ERRNB  <test>          ;;Fehler, wenn mehr
                                ;;als ein Parameter
          .
          .
          .
        ENDM
```

In diesem Beispiel werden die bedingten Fehlerpseudooperationen benutzt, um zu sichern, dass ein und nur ein Parameter an den

Makro uebergeben wird.

.ERRB erzeugt einen Fehler, wenn kein Parameter uebergeben wurde. .ERRNB generiert einen Fehler bei der Uebergabe von mehr als einem Parameter.

7.3.5. .ERRIDN und .ERRDIF

Syntax:

```
.ERRIDN <zeichenkette1>,<zeichenkette2>
.ERRDIF <zeichenkette1>,<zeichenkette2>
```

Diese Pseudooperationen pruefen zwei Zeichenketten auf Identitaet. .ERRIDN erzeugt einen Fehler, wenn die Zeichenketten identisch, .ERRDIF, wenn sie verschieden sind.

Die Zeichenketten koennen Namen, Zahlen oder Ausdruecke sein. Zwei Zeichenketten sind identisch, wenn jedes Zeichen aus <zeichenkette1> gleich dem entsprechenden Zeichen aus <zeichenkette2> ist. Bei der Zeichenpruefung werden Gross- und Kleinschreibung beruecksichtigt. **Die spitzen Klammern (<>) sind erforderlich.**

Beispiel:

```
summe   MACRO   ad1,ad2,sum
        .ERRIDN <ax>,<ad2>      ;;Fehler, wenn ad2='ax'
        .ERRIDN <AX>,<ad2>      ;;Fehler, wenn ad2='AX'
        mov     ax,ad1          ;;wuerde ueberschrieben,
                                ;;wenn ad2=ax waere
        add     ax,ad2
        mov     sum,ax          ;;sum muss ein Register
                                ;;oder Speicherplatz
                                ;;sein
        ENDM
```

.ERRIDN wird hier verwendet, um das AX-Register gegen die Uebergabe als 2. Parameter zu schuetzen, weil der Makro sonst fehlerhaft arbeiten wuerde. Es ist zu beachten, dass die .ERRIDN-Anweisung zweimal geschrieben wird, um die beiden wahrscheinlichen Schreibweisen fuer das Register zu testen.

8. Makroprogrammierung

8.1. Einleitung

Dieses Kapitel erklaert, wie Makros in einer Quelldatei erzeugt und verwendet werden. Es werden die Makropseudooperationen und die speziellen Makrooperatoren besprochen. Da Makros mit bedingten Pseudooperationen eng verwandt sind, ist das Studium des Kapitels 7 zum Verstaendnis einiger Beispiele in diesem Kapitel notwendig.

Die Pseudooperationen fuer die Makroprogrammierung erlauben es, einen benannten Block von Quellanweisungen zu schreiben und dann diesen Namen stellvertretend fuer die Anweisungen in der Quelldatei zu verwenden. Waehrend der Uebersetzung ersetzt MASM jedes Auftreten des Makronamens durch die Anweisungen in der Makrodefinition. Solche Makrodefinitionen koennen ueberall in der Quelldatei und in jeder Anzahl stehen. Der Makroname wird dann an jeder Stelle eingefuegt, an der der Makroblock uebersetzt werden soll. **Die Makrodefinition muss immer vor dem ersten Aufruf dieses Makros stehen.** Parameter koennen an einen Makro uebergeben werden.

Makros koennen in getrennten Dateien abgelegt sein. Sie werden durch die Pseudooperation INCLUDE fuer das Programm verfuegbar gemacht (siehe dazu den Abschnitt 9.2.).

Oft kann eine Aufgabe sowohl durch eine Prozedur als auch durch einen Makro geloest werden. Beispielsweise realisiert die Prozedur addup aus dem Abschnitt 3.10. das Gleiche, wie der Makro addup im Abschnitt 8.2.1. Makros werden bei jedem Auftreten des Makronamens erweitert. So kann sich durch wiederholten Aufruf die Laenge einer ausfuehrbaren Datei vergroessern. Prozeduren nehmen weniger Platz ein. Durch den Aufwand des Rettens und Wiederherstellens von Adressen und Parametern sind sie langsamer.

8.2. Makro-Pseudooperationen

Makro-Pseudooperationen sind:

Pseudo-operation	Bedeutung
MACRO	Anfang einer Makrodefinition
ENDM	Ende einer Makrodefinition oder eines Wiederholungsblockes
LOCAL	Generieren von Marken innerhalb eines Makros
PURGE	Loeschen von Makrodefinitionen
REPT **IRP** **IRPC**	Beginn eines Wiederholungsblockes
EXITM	Vorzeitiges Beenden einer Makroerweiterung oder eines Wiederholungsblockes

Die Pseudooperationen MACRO und ENDM bezeichnen den Anfang und das Ende eines Makroblockes. Durch LOCAL werden Marken definiert, die nur innerhalb eines Makros verwendet werden. PURGE

loescht vorher definierte Makros. Mit EXITM kann ein Makro vorzeitig verlassen werden, bevor alle Anweisungen dieses Blokkes erweitert worden sind.

REPT, IRP und IRPC erzeugen aneinander grenzende Bloecke von sich wiederholenden Anweisungen. Diese Wiederholungsbloecke werden haeufig innerhalb von Makros eingesetzt. Sie koennen ebenso selbstaendig verwendet werden. Die Anzahl der Wiederholungen kann durch die Angabe einer Zahl gesteuert werden, oder der Block wird fuer jeden Parameter in einer Liste bzw. fuer jedes Zeichen in einer Zeichenkette einmal wiederholt.

8.2.1. MACRO und ENDM

Syntax:

```
<name> MACRO [<dummyparameter>[,<dummyparameter>,...]]
       <anweisungen>
       ENDM
```

MACRO und ENDM erzeugen einen Makro mit dem Namen <name>, der die gegebenen <anweisungen> enthaelt.

<name> muss ein gueltiger Name sein und darf nur einmal vorkommen. Er wird in der Quelldatei zum Aufruf des Makros benutzt. <dummyparameter> sind symbolische Parameter, die durch die echten beim Makroaufruf uebergebenen Werte durch Eins-zu-Eins-Textsubstitution ersetzt werden. Mehrere Parameter muessen durch Kommas (,) getrennt werden.

<anweisungen> koennen alle gueltigen MASM-Anweisungen, einschliesslich andere Makro-Pseudooperationen sein. Die Anweisungsanzahl ist beliebig. Es koennen beliebig viele symbolische Parameter beliebig oft in diesen Anweisungen verwendet werden.

Ein Makro wird in dem Moment aufgerufen, wenn sein Name in einer Quelldatei erscheint (Makronamen in Kommentaren werden ignoriert). MASM kopiert die Anweisungen der Makrodefinition an die Stelle des Aufrufs. Dabei wird jeder symbolische Parameter in diesen Anweisungen durch den im Aufruf uebergeben aktuellen Wert ersetzt.

Makrodefinitionen koennen geschachtelt werden. Das bedeutet, ein Makro kann innerhalb eines anderen Makros definiert sein. MASM behandelt innere Definitionen solange nicht, bis der aeussere Makro aufgerufen worden ist. Deshalb kann ein innerer Makro nicht aufgerufen werden solange der aeussere Makro nicht mindestens einmal aufgerufen wurde. Makrodefinitionen koennen in beliebiger Tiefe geschachtelt werden. Die Schachtelungstiefe ist nur von der Speichergroesse waehrend der Uebersetzung abhaengig.

Makrodefinitionen koennen Aufrufe anderer Makros beinhalten. Die inneren Makroaufrufe werden wie jeder andere Makroaufruf erweitert, aber nur dann, wenn der aeussere Makro aufgerufen wurde.

Makrodefinitionen koennen rekursiv sein, das heisst sie koennen sich selbst aufrufen, wie in dem Beispiel im Abschnitt 7.2.4. gezeigt wurde.

Beispiel:

```
addup   MACRO   ad1,ad2,ad3
        mov     ax,ad1          ;;1. Parameter -> AX
        add     ax,ad2          ;;die naechsten 2 Parameter
        add     ax,ad3          ;;addieren und die Summe in
        ENDM                    ;;AX belassen
```

Das vorangehende Beispiel definiert einen Makro mit dem Namen addup. Er verwendet 3 symbolische Parameter zum Addieren von 3 Worten. Ihre Summe wird im Register AX belassen. Die 3 symbolischen Parameter werden beim Aufruf des Makros durch die aktuellen Werte ersetzt.

MASM uebersetzt die Anweisungen in dem Makro nur, wenn er aufgerufen worden ist und nur an der Stelle in der Quelldatei, an der der Makroaufruf steht. Deshalb sind alle Adressen im uebersetzten Kode relativ zum Makroaufruf und nicht zur Makrodefinition. Die Makrodefinition selbst wird nicht uebersetzt, sie erzeugt keinen Kode.

Man muss bei der Verwendung des Wortes MACRO nach den Pseudooperationen TITLE, SUBTTL und NAME vorsichtig sein. Das wuerde den Assembler veranlassen, eine Makrodefinition mit dem Namen TITLE, SUBTTL oder NAME zu beginnen. Mit der Zeile

```
TITLE Macro Datei
```

sollte einer "INCLUDE-Datei" der Titel "Macro Datei" geben werden. Diese Anweisung hat aber zur Folge, dass ein Makro genannt TITLE erzeugt und Datei als symbolischer Parameter angesehen wird. Weil kein zugehoeriges ENDM gefunden wird, generiert MASM einen Fehler.

Um dieses Problem zu vermeiden, sollte man das Wort MACRO anders schreiben, wenn es als Name oder Titel verwendet werden soll, z. B. MAKRO oder _MACRO.

Beachte:

In einem Makro wird jedes Auftreten eines symbolischen Parameters ersetzt, auch wenn dies nicht beabsichtigt war. Wurde zum Beispiel ein Registername, wie AX oder BL, als symbolischer Parameter benutzt, ersetzt MASM diesen Registernamen bei der Makroerweiterung immer durch den uebergebenen Wert. Enthaelt der Makro Anweisungen, die diese Register verwenden, wird der Makro fehlerhaft erweitert.

Beachte:

Makros koennen wiederholt definiert werden. Der urspruengliche Makro ist vorher nicht unbedingt zu loeschen. Die neue Definition ersetzt automatisch die vorhergehende. Wenn ein Makro in sich selbst neu definiert wird, muss gesichert werden, dass zwischen den beiden ENDM-Anweisungen keine weiteren Zeilen stehen. Das folgende Beispiel erzeugt fehlerhaften Kode:

Beispiel:

```
beisp   MACRO
          .
          .
          .
        beisp   MACRO
                  .
                  .
                  .
                ENDM
        ;;Kommentare oder Anweisungen sind nicht erlaubt
        ENDM
```

Um den Fehler zu beseitigen, ist die Zeile zwischen den ENDM-Anweisungen zu entfernen.

8.2.2. Makroaufrufe

Syntax:

```
<name> [<aktueller wert>[,<aktueller wert>,...]]
```

Ein Makroaufruf veranlasst den Assembler, die Anweisungen des Makros <name> an die Aufrufstelle zu kopieren und die symbolischen Parameter in diesen Anweisungen durch die entsprechenden aktuellen Werte zu ersetzen. <name> muss der Name einer Makrodefinition sein, die **vorher** in der Quelldatei steht.

<aktueller wert> kann

- **jeder Name**
- **eine Zahl**
- **ein anderer Wert**

sein. Es sind beliebig viele aktuelle Werte moeglich. Sie muessen alle auf eine Zeile passen. Mehrere Werte muessen durch Kommas (,), Tabulatoren oder Leerzeichen voneinander getrennt werden.

MASM ersetzt den ersten symbolischen Parameter durch den ersten aktuellen Wert, den zweiten durch den zweiten usw. Besitzt ein Makroaufruf mehr aktuelle Werte als symbolische Parameter, werden die ueberzaehligen ignoriert. Hat er weniger aktuelle Werte als symbolische Parameter, so wird jeder restliche Parameter

durch eine Null- bzw. Leer-Kette ersetzt. Die Pseudooperationen IFB, IFNB, und .ERRB koennen verwendet werden, um die Makros auf Nullketten zu testen und angemessen zu reagieren. Siehe dazu die Abschnitte 7.2.4. und 7.3.4.

Soll eine Liste von aktuellen Werten als ein einziger aktueller Wert uebergeben werden, muss diese Liste in spitze Klammern (<>) eingeschlossen werden. Die einzelnen Werte in der Liste sind durch Kommas (,) zu trennen.

Beispiele:

```
zuweis  1,2,3,4,5
```

Das erste Beispiel uebergibt 5 numerische Werte an den Makro zuweis.

```
zuweis  <1,2,3,4,5>
```

In diesem Beispiel wird nur ein Wert an den Makro zuweis uebergeben. Dieser Wert besteht aus einer Liste von 5 Zahlen.

```
addup   bx,2,count
```

Im letzten Beispiel werden 3 Werte an den Makro addup uebergeben. MASM ersetzt die zugehoerigen symbolischen Parameter exakt so, wie die Parameter im Makroaufruf geschrieben sind. Angenommen, addup waere der am Ende des Abschnitts 8.2.1. definierte Makro, dann wuerde der Assembler folgenden Kode erweitern:

```
mov     ax,bx
add     ax,2
add     ax,count
```

Im Abschnitt 10.4. ist ein Beispiel gegeben, wie Makros in der Assemblerliste dargestellt werden.

8.2.3. LOCAL

Syntax:

```
LOCAL <dummyname>[,<dummyname>,...]
```

Die Pseudooperation LOCAL erzeugt einen einmal vorkommenden Symbolnamen fuer die Verwendung in Makros. <dummyname> ist ein symbolischer Name, der bei der Makroerweiterung durch einen nur einmal vorkommenden Namen ersetzt wird. Es ist mindestens ein <dummyname> erforderlich. Werden mehrere angegeben, muessen sie durch Kommas (,) getrennt werden. <dummyname> kann in jeder Anweisung innerhalb des Makros verwendet werden.

Bei jeder Makroerweiterung erzeugt MASM einen neuen aktuellen Namen fuer den symbolischen Namen in folgender Form:

??<nummer>

<nummer> ist eine Hexadezimalzahl in den Grenzen von 0000 bis FFFF. Symbole dieses Formats duerfen nicht anderweitig verwendet werden, weil dann Marken oder Symbole mehrfach auftreten koennen. In der Liste erscheinen in der Makrodefinition die symbolischen Namen, aber in jeder Makroerweiterung wird der aktuelle Name eingesetzt.

Die LOCAL-Anweisung findet nur innerhalb von Makros Anwendung. Enthaelt ein Makro Marken oder Symbole und wird er in einer Quelldatei mehr als einmal aufgerufen, entsteht normalerweise ein Fehler. Dieser zeigt an, dass die Marke oder das Symbol mehrfach definiert wurde, weil dieselbe Marke bzw. dasselbe Symbol in jeder Makroerweiterung auftritt. Vermieden wird dies, indem alle innerhalb von Makros verwendeten Marken und Symbole mit der Pseudooperation LOCAL vereinbart werden.

Beachte:

Die Pseudooperation LOCAL kann nur in einer Makrodefinition verwendet werden. Sie muss vor allen anderen Anweisungen in dieser Makrodefinition stehen. Eine Kommentarzeile oder ein Befehl vor LOCAL verursacht eine Warnung.

Beispiel:

```
expo    MACRO   faktor,exponent
        LOCAL   wieder,gonull ;;Vereinbaren Symbole
                              ;;fuer Makro
        mov     cx,exponent   ;;exponent ist Zaehler
                              ;;fuer Schleife
        mov     ax,1          ;;Multiplizieren mit 1
                              ;;das erste Mal
        jcxz    gonull        ;;beenden, bei exponent=0
        mov     bx,faktor
wieder: mul     bx            ;;Multiplizieren bis Ende
        loop    wieder
gonull:
        ENDM
```

In diesem Beispiel definiert die Pseudooperation LOCAL die symbolischen Namen wieder und gonull. Diese Namen werden bei jeder Makroerweiterung durch nur einmal vorkommende Namen ersetzt. Beim ersten Mal erhaelt wieder den Namen ??0000 und gonull wird ??0001 zugewiesen. Beim zweiten Aufruf heisst wieder ??0002 und gonull ??0003 usw.

8.2.4. PURGE

Syntax:

PURGE <makroname>[,<makroname>,...]

Die Pseudooperation PURGE loescht die aktuelle Definition des Makros mit dem Namen <makroname>. Jeder spaetere Aufruf dieses Makros fuehrt bei der Uebersetzung zu einem Fehler.

Die PURGE-Anweisung wurde zum Loeschen nicht benoetigter Makros aus dem Speicher vorgesehen. Ist <makroname> ein Befehl oder die Mnemonik einer Pseudooperation, dann ist ihre vorherige Bedeutung wieder hergestellt.

Die PURGE-Anweisung wird haeufig im Zusammenhang mit "Makrobibliotheken" benutzt, um nur die Makros aus der Bibliothek auszuwaehlen, die wirklich benoetigt werden. Eine Makrobibliothek ist eine Datei, die Makrodefinitionen enthaelt. Sie wird der Quelldatei durch eine INCLUDE-Anweisung hinzugefuegt. Dann werden mit der PURGE-Anweisung die nicht benoetigten Definitionen entfernt.

Es ist nicht noetig, einen Makro mit PURGE zu loeschen bevor er neu definiert werden soll. Jede neue Definition eines Makros loescht automatisch die vorhergehende. Ein Makro kann sich selbst loeschen, sobald eine PURGE-Anweisung auf der letzten Zeile des Makros steht. Dieser Makro kann dann nur einmal aufgerufen werden!

Beispiele:

1. **PURGE addup**

Das erste Beispiel loescht den Makro addup.

2. **PURGE mak1,mak2,mak9**

Im zweiten Beispiel werden die Makros mak1, mak2 und mak9 entfernt.

8.2.5. REPT und ENDM

Syntax:

```
REPT    <ausdruck>
        <anweisungen>
ENDM
```

Die Pseudooperationen REPT und ENDM schliessen einen Block von Anweisungen ein, der sooft wiederholt wird, wie <ausdruck> angibt. Ausdruck muss eine vorzeichenlose 16-Bit-Zahl ergeben. Er darf keine externen oder undefinierten Symbole enthalten. <anweisungen> koennen alle gueltigen Anweisungen sein.

Beispiel:

```
X       =       0
        REPT    10
X       =       X+1
        DB      X
        ENDM
```

Dieses Beispiel wiederholt die Anweisungen = und DB zehnmal. Das Ergebnis sind 10 Datenbytes mit den Werten 1 bis 10.

8.2.6 IRP und ENDM

Syntax:

```
IRP <symbolischer name>,<[<parameter>[,<parameter>,...]]>
        <anweisungen>
ENDM
```

Die Pseudooperationen IRP und ENDM schliessen einen Block von Anweisungen ein, der fuer jeden Parameter in den spitzen Klammern (<>) wiederholt wird. <symbolischer name> ist der Name eines symbolischen Parameters, der durch die aktuellen <parameter> ersetzt wird.

<parameter> kann

- **jedes gueltige Symbol**
- **eine Zeichenkette**
- **eine Zahl**
- **eine Zeichenkonstante**

sein. Es koennen beliebig viele Parameter angegeben werden. Mehrere Parameter sind durch Kommas (,) voneinander zu trennen. Die spitzen Klammern (<>) um die Parameterliste sind erforderlich. <anweisungen> koennen alle gueltigen Assembleranweisungen sein. <symbolischer name> kann ueberall in diesen Anweisungen geschrieben werden.

Trifft MASM auf eine IRP-Anweisung, legt er fuer jeden Parameter einer durch spitze Klammern (<>) eingeschlossenen Liste eine Kopie der Anweisungen an. Waehrend des Kopierens ersetzt er in den Anweisungen jedesmal <symbolischer name> durch den aktuellen <parameter>. Wird ein Nullparameter in der Liste gefunden, dann wird der symbolische Parameter durch einen Nullwert ersetzt. Bei leerer Parameterliste wird die IRP-Anweisung ignoriert. Es werden keine Anweisungen kopiert.

Beispiel:

```
IRP     x,<0,1,2,3,4,5,6,7,8,9>
        DB      10 DUP(x)
ENDM
```

In diesem Beispiel wird die DB-Anweisung zehnmal wiederholt, dabei werden die Zahlen in der Liste in jeder Wiederholung eingesetzt. Die resultierenden Anweisungen erzeugen 100 Datenbytes mit jeweils zehn Gruppen der Werte 0 bis 9.

Beachte:

Angenommen, die IRP-Anweisung wird innerhalb einer Makrodefinition benutzt und die Parameterliste ist demzufolge ein symbolischer Parameter des Makros. In diesem Fall muss der symbolische Parameter in spitze Klammern eingeschlossen werden. In dem folgenden Makro wird beispielsweise der symbolische Parameter x als Parameterliste fuer die IRP-Anweisung verwendet.

Beispiel:

```
zuweis  MACRO   x
        IRP     y,<x>
        DB      y
        ENDM
        ENDM
```

Wird dieser Makro aufgerufen durch

```
zuweis  <0,1,2,3,4,5,6,7,8,9>
```

entstehlt folgende Makroerweiterung

```
IRP     y,<0,1,2,3,4,5,6,7,8,9>
DB      y
ENDM
```

Der Makro entfernt die Klammern von dem aktuellen Parameter, bevor der symbolische Parameter ersetzt wird. Man muss die spitzen Klammern um die Parameterliste selbst vorsehen.

8.2.7. IRPC und ENDM

Syntax:

```
IRPC    <symbolischer name>,<zeichenkette>
        <anweisungen>
ENDM
```

Diese Pseudooperationen IRPC und ENDM schliessen einen Block von Anweisungen ein, der fuer jedes Zeichen in <zeichenkette> wiederholt wird. <symbolischer name> ist ein symbolischer Parame-

ter, der durch das aktuelle Zeichen in der Zeichenkette ersetzt wird. <zeichenkette> kann jede Kombination von Buchstaben, Ziffern und anderen Zeichen sein. Die Zeichenkette sollte in spitze Klammern eingeschlossen werden, wenn sie Trennzeichen enthaelt. Der symbolische Parameter kann beliebig oft in diesen Anweisungen verwendet werden.

Gelangt MASM an eine IRPC-Anweisung, legt er fuer jedes Zeichen der Kette eine Kopie der Anweisungen ab. Waehrend dieses Vorganges wird jedesmal der symbolische Parameter durch das aktuelle Zeichen ersetzt.

Beispiel:

```
IRPC    x,0123456789
        DB      x+1
ENDM
```

In diesem Beispiel wird die DB-Anweisung zehnmal wiederholt fuer jedes Zeichen in der Kette 0123456789. Die daraus resultierenden Anweisungen erzeugen 10 Datenbytes mit den Werten von 1 bis 10.

8.2.8. EXITM

Syntax:

```
EXITM
```

Die Pseudooperation EXITM teilt dem Assembler mit, dass die Erweiterung des Makros oder Wiederholungsblockes beendet werden soll. Die Uebersetzung wird mit der naechsten Anweisung nach dem Makroaufruf oder dem Wiederholungsblock fortgesetzt. Die EXITM-Anweisung wird haeufig in Verbindung mit der IF-Anweisung benutzt, um die bedingte Erweiterung der letzten Anweisungen in einem Makro oder Wiederholungsblock zu ermoeglichen.

Beim Auftreten von EXITM verlaesst der Assembler den Makro oder Wiederholungsblock sofort. Alle uebrigen Anweisungen werden nicht mehr verarbeitet. EXITM in einem inneren (geschachtelten) Makro oder Wiederholungsblock bewirkt die Rueckkehr zur Erweiterung des aeusseren Blockes.

Beispiel:

```
zuweis  MACRO   wieder
        X       =       0
        REPT    wieder          ;;Wiederholen bis 256 *
                IFE     X-0FFh  ;;X = 255 ?
                EXITM           ;;wenn so, dann beenden
                ELSE
                DB      X       ;;andernfalls X zuweisen
                ENDIF
        X       =       X+1     ;;X erhoehen
        ENDM
        ENDM
```

Dieses Beispiel definiert einen Makro, der nicht mehr als 255 Datenbytes erzeugt. Der Makro enthaelt eine IFE-Anweisung zur Pruefung des Ausdrucks X-0FFh. Ist der Ausdruck 0 (X = 255), wird die EXITM-Anweisung abgearbeitet und die Erweiterung des Makros beendet.

8.3. Makro-Operatoren

In Makros und bedingten Pseudooperationen koennen folgende spezielle Operatoren verwendet werden:

Operator	Bedeutung
&	Ersetzungsoperator
<>	Literal-Text-Operator
!	Literal-Zeichen-Operator
%	Ausdruck-Operator
;;	Makrokommentar

Diese Operatoren fuehren bei Verwendung in Makrodefinitionen oder bei bedingter Uebersetzung spezielle Steueroperationen aus (z. B. Textsubstitution). Sie sind in den Abschnitten 8.3.1. bis 8.3.5. beschrieben.

8.3.1. Ersetzungsoperator

Syntax:

```
&<symbolischer parameter>
oder
<symbolischer parameter>&
```

Der Ersetzungsoperator (&) zwingt MASM den <symbolischen para-

meter> durch seinen zugehoerigen aktuellen Wert zu ersetzen. Dieser Operator wird verwendet, wenn einem symbolischen Parameter unmittelbar andere Zeichen vorausgehen oder folgen, oder wenn der Parameter in einer in Hochkommas eingeschlossenen Zeichenkette auftritt.

Beispiel:

```
errgen  MACRO   Y,X
error&X DB      'Error &Y - &X'
        ENDM
```

Im obigen Beispiel ersetzt MASM beim Aufruf des Makros errgen die symbolischen Parameter &X und &Y durch die uebergebenen aktuellen Werte. Wird der Makro mit der Anweisung

```
errgen  1,wait
```

aufgerufen, entsteht folgende Erweiterung:

```
errorwait       DB      'Error 1 - wait'
```

Beachte:

Bei geschachtelten Makros koennen zusaetzliche Ampersands (&) verwendet werden, um das aktuelle Ersetzen von symbolischen Parametern zu verzoegern. Im Allgemeinen muessen so viele Ampersands eingefuegt werden, wie die Schachtelungstiefe betraegt.

In der folgenden Makrodefinition wird der Ersetzungsoperator beispielsweise zweimal bei Z verwendet. Dies sichert, dass er erst waehrend der Abarbeitung der IRP-Anweisung ersetzt wird:

```
zuweis  MACRO   X
        IRP     Z,<1,2,3>
        X&&Z    DB      Z
        ENDM
        ENDM
```

In diesem Beispiel wird der Parameter X unmittelbar beim Makroaufruf ersetzt. Der symbolische Parameter Z wird solange nicht ersetzt, bis die IRP-Anweisung abgearbeitet wird. Das bedeutet, der Parameter wird fuer jede Zahl in der IRP-Parameterliste einmal ersetzt.

Der Makro aufgerufen mit

```
zuweis  var
```

generiert folgende Anweisungen:

```
var1    DB      1
var2    DB      2
var3    DB      3
```

8.3.2. Literal-Text-Operator

Syntax:

```
<text>
```

Der Literal-Text-Operator veranlasst MASM text als ein einziges Literal zu behandeln. Das ist unabhaengig davon, ob er Kommas, Leerzeichen oder andere Trennzeichen enthaelt. Dieser Operator findet oft beim Aufruf von Makros oder in IRP-Anweisungen Verwendung, wenn die Werte in einer Parameterliste als ein einziger Parameter behandelt werden sollen.

Ausserdem zwingt der Literal-Text-Operator MASM dazu, spezielle Zeichen, wie das Semikolon (;) oder das Ampersand (&), nicht in ihrer speziellen Bedeutung, sondern als Zeichen zu behandeln. Beispielsweise wird ein Semikolon in spitzen Klammern <;> als das Zeichen Semikolon und nicht als Einleitung eines Kommentars gedeutet.

MASM beseitigt immer dann ein Paar spitze Klammern, wenn der Parameter in einem Makro verwendet wird. Bei der Anwendung in geschachtelten Makros muessen so viele spitze Klammern gesetzt werden, wie die Schachtelungstiefe betraegt.

8.3.3. Literal-Zeichen-Operator

Syntax:

```
!<zeichen>
```

Dieser Operator bewirkt, dass <zeichen> als Literalzeichen vom Assembler behandelt wird. Dieser Operator ist notwendig, wenn Zeichen mit spezieller Bedeutung (z. B. Semikolon (;) oder Ampersand (&)) nicht in dieser Bedeutung, sondern als Zeichen verwendet werden sollen.

8.3.4. Ausdruck-Operator

Syntax:

```
%<text>
```

Der Operator % veranlasst den Assembler <text> als Ausdruck zu bewerten. MASM berechnet den Wert des Ausdruckes unter Verwendung der aktuellen Zahlenbasis und ersetzt <text> durch diesen neuen Wert. <text> muss einen gueltigen Ausdruck darstellen.

Der Ausdruck-Operator wird in Makroaufrufen verwendet, wo der Programmierer das Ergebnis eines Ausdrucks statt des Ausdrucks uebergeben muss.

Beispiel:

```
druck   MACRO   meld,num
        IF2                     ;;nur im Pass 1
        %OUT    #&meld&num      ;;Anzeigen Meldung und
                                ;; Zahl auf Bildschirm
        ENDIF
        ENDM

sym1    EQU     100
sym2    EQU     200

        druck   <sym1+sym2=>,%(sym1+sym2) ;Makroaufruf
```

In diesem Beispiel uebergibt der Makroaufruf das Textliteral **sym1+sym2=** an den symbolischen Parameter **meld.** Er uebergibt den Wert 300 (das Ergebnis des Ausdruckes sym1+sym2) an den symbolischen Parameter **num.** Im Ergebnis zeigt MASM beim Erreichen des Makroaufrufs waehrend der Uebersetzung die Meldung **sym1+sym2=300** auf dem Bildschirm an.

Die Pseudooperation %OUT, die diese Meldung auf den Bildschirm bringt, ist im Abschnitt 9.4. beschrieben und die Pseudooperation IF2 im Abschnitt 7.2.2.

8.3.5. Makro-Kommentar

Syntax:

```
;;<text>
```

Ein Makrokommentar ist jeder Text in einer Makrodefinition, der nicht in die Makroerweiterung uebernommen werden soll. Jeder Text, der auf zwei Semikolons (;;) folgt, wird vom Assembler ignoriert. Er erscheint in der Assemblerliste nur einmal in der Makrodefinition.

Der normale Kommentaroperator (;) kann in Makros auch verwendet werden. Aber dieser Kommentar wird in der Assemblerliste in die Makroerweiterung mit uebernommen. Ob normaler Kommentar in Makroerweiterungen aufgelistet wird, haengt von der Verwendung der Pseudooperation .LALL, .XALL und .SALL ab (siehe Abschnitt 9.11.).

9. Pseudooperationen zur Dateisteuerung

9.1. Einleitung

In diesem Kapitel werden die Pseudooperationen zur Dateisteuerung von MASM beschrieben. Sie ermoeglichen die Steuerung von Quell-, Objektkode- oder Listendateien.

Zu dieser Gruppe gehoeren folgende Pseudooperationen:

Pseudo-operation	Bedeutung
INCLUDE	Eine Quelldatei einfuegen
.RADIX	Aendern der Standardeingabezahlenbasis
%OUT	Eine Meldung auf dem Bildschirm anzeigen
NAME	Name fuer die Objektkodedatei
TITLE	Titelzeile fuer die Assemblerliste setzen
SUBTTL	Untertitel fuer die Assemblerliste setzen
PAGE	Seitenlaenge und Zeilenbreite fuer Assemblerliste festlegen
.LIST	Anweisungen in die Assmeblerliste aufnehmen
.XLIST	Unterdruecken der Anweisungen in der Assemblerliste
.LFCOND	Auflisten der falschen Bedingungen
.SFCOND	Unterdruecken des Auflistens der falschen Bedingungen
.TFCOND	Umkippen der Listensteuerung falscher Bedingungen
.LALL	Anweisungen, die durch Makroerweiterung entstanden sind, in die Assemblerliste aufnehmen
.SALL	Anweisungen, die durch Makroerweiterung entstanden sind, in der Assemblerliste unterdruecken
.XALL	Kommentare aus der Makroliste ausschliessen
.CREF	Symbole in Crossreferenzdatei aufnehmen
.XCREF	Unterdruecken Symbolliste

9.2. INCLUDE

Syntax:

INCLUDE <dateiname>

Die INCLUDE-Anweisung fuegt Quellkode aus der durch <dateiname> gegebenen Quelldatei in die aktuelle Quelldatei waehrend der Uebersetzung ein. <dateiname> muss der Name einer existierenden Datei sein. Wenn sich die Datei nicht im aktuellen Arbeitslaufwerk befindet, wird ein vollstaendiger oder unvollstaendiger Pfadname angegeben.

MASM sucht die "INCLUDE-Datei" (Quelldatei, die durch <dateiname> spezifiziert wurde) zuerst in den Pfaden, die mit dem /I-Schalter festgelegt wurden. Dann wird das aktuelle Verzeichnis geprueft. Wird die genannte Datei nicht gefunden, gibt der Assembler eine Fehlermeldung aus und haelt an.

Gelangt der Assembler an eine INCLUDE-Anweisung, eroeffnet er die spezifizierte Quelldatei und uebersetzt sofort ihre Anweisungen. Nachdem MASM alle Anweisungen gelesen hat, setzt er die Uebersetzung mit der auf die Pseudooperation INCLUDE folgenden Anweisung fort.

Geschachtelte INCLUDE-Anweisungen sind erlaubt. Das heisst eine in einer INCLUDE-Anweisung enthaltene Datei kann wieder eine INCLUDE-Anweisung enthalten. Die aus einer "INCLUDE-Datei" stammenden Anweisungen werden in der Liste mit dem Buchstaben C gekennzeichnet.

Verzeichnisse koennen im Pfadnamen der INCLUDE-Anweisung entweder durch den Schraegstrich (/) oder den inversen Schraegstrich (\) eingeleitet werden.

Es sollte nur der Dateiname, aber kein Pfadname in einer INCLUDE-Anweisung spezifiziert werden, wenn mit dem /I-Schalter ein Suchpfad festgelegt werden soll. Der /I-Schalter ist im Abschnitt 10.3.6. erklaert.

Beispiele:

```
INCLUDE entry                  ;Dateiname
INCLUDE b:\include\record      ;Pfadname
INCLUDE /include/as/stdio      ;Pfadname
INCLUDE localinc\define.inc    ;unvollstaendiger Pfadname
```

9.3. .RADIX

Syntax:

.RADIX <ausdruck>

Mit der Pseudooperation .RADIX wird die Eingabezahlenbasis fuer Zahlen in der Quelldatei gesetzt. <ausdruck> ist eine Zahl in den Grenzen zwischen 2 und 16. Er definiert, ob die Zahlen binaer, oktal, dezimal, hexadezimal oder zu einer anderen Basis sind.

Die gebraeuchlichsten Zahlenbasen sind:

Basis	Zahlentyp
2	Binaer
8	Oktal
10	Dezimal
16	Hexadezimal

<ausdruck> wird unabhaengig von der aktuellen Eingabezahlenbasis immer als Dezimalzahl betrachtet. Der Standard fuer die Eingabezahlenbasis ist dezimal.

Beachte:

.RADIX hat keine Wirkung auf die Pseudooperationen DD, DQ oder DT. Die im Ausdruck dieser Pseudooperationen eingetragenen Zahlen werden immer als dezimal bewertet, wenn nicht eine spezielle Zahlenbasis an den Wert angehaengt ist.

.RADIX hat auch keine Wirkung auf die wahlfreie Spezifikation der Zahlenbasis B und D, die bei ganzen Zahlen verwendet wird. Tritt der Buchstabe B oder D am Ende einer ganzen Zahl auf, so wird das immer als Spezifikation einer Zahlenbasis betrachtet, auch wenn die Eingabezahlenbasis 16 ist.

Die Zahl 0ABCD wird beispielsweise als dezimal interpretiert, wenn die aktuelle Zahlenbasis 16 ist, also als ungueltige Zahl und nicht wie beabsichtigt als Hexadezimalzahl 0ABCD. Es ist 0ABCDh zu schreiben, um diese Zahl als hexadezimal zu kennzeichnen. Aehnlich wird 11B als 11 binaer behandelt, eine gueltige Zahl, aber es war 11B hexadezimal beabsichtigt.

Beispiele:

```
.RADIX 16
.RADIX 2
```

Das erste Beispiel definiert die aktuelle Eingabezahlenbasis hexadezimal, das zweite Beispiel binaer.

9.4. %OUT

Syntax:

%OUT [<text>]

Erreicht der Assembler waehrend der Uebersetzung die %OUT-Anweisung, zeigt er den spezifizierten <text> auf dem Bildschirm an. Diese Pseudooperation gestattet bei langen Uebersetzungen an bestimmten Punkten Meldungen anzuzeigen.

%OUT generiert in beiden Paessen eine Ausgabe. Mit IF1 und IF2 kann die Ausgabe auf den entsprechenden Pass begrenzt werden.

Beispiel:

```
IF1
        %OUT Erster Pass beendet
ENDIF
```

Diese Beispielanweisungen koennten am Ende einer Quelldatei stehen. Die Meldung **Erster Pass beendet** wuerde dann am Ende des ersten Passes angezeigt und im zweiten Pass ignoriert.

9.5. NAME

Syntax:

NAME <modulname>

Die Pseudooperation NAME gibt dem aktuellen Modul den Namen <modulname>. Dieser Modulname wird vom Programmverbinder benutzt, wenn Fehlermeldungen angezeigt werden.

<modulname> kann jede Kombination von Buchstaben und Ziffern sein. Die Laenge des Namens ist beliebig, aber es werden nur die ersten sechs Zeichen verwendet. Der Name darf nur einmal vorkommen und kein reserviertes Wort sein.

Wurde keine NAME-Anweisung verwendet, bildet der Assembler einen Standardmodulnamen aus den ersten sechs Zeichen des in der TITLE-Anweisung angegebenen Textes. Wurde auch keine TITLE-Anweisung gefunden, so ist der Standardname A.

Beispiel:

```
NAME Grafik
```

Dieses Beispiel setzt den Modulnamen auf Grafik.

9.6. TITLE

Syntax:

TITLE <text>

Die Pseudooperation TITLE spezifiziert die Titelzeile fuer die Assemblerliste. Sie wird auf der 1. Zeile jeder Seite ausgeben. Als <text> ist jede Zeichenkombination bis zu einer Laenge von 60 Zeichen erlaubt.

Je Modul darf nur eine TITLE-Anweisung angegeben werden. Die ersten 6 vom Leerzeichen verschiedenen Zeichen werden als Modulname verwendet, wenn der Modul keine NAME-Anweisung enthaelt.

Beispiel:

TITLE Grafik - Erstes Programm

Dieses Beispiel spezifiziert "Grafik - Erstes Programm" als Titelzeile. Wenn der Modul keine NAME-Anweisung enthaelt, dann heisst der Modul Grafik (die ersten 6 Zeichen der Titelzeile).

9.7. SUBTTL

Syntax:

SUBTTL [<text>]

Die Pseudooperation SUBTTL spezifiziert einen Untertitel fuer die Liste. Diese Zeile wird auf jeder Seite unmittelbar nach der Titelzeile ausgegeben. Der <text> kann jede Zeichenkombination bis zu einer Laenge von 60 Zeichen sein. Ist kein Text angegeben, bleibt die Zeile leer.

Es koennen beliebig viele SUBTTL-Anweisungen in einem Programm angegeben werden. Eine neue SUBTTL-Anweisung ersetzt den Text der vorangegangen.

Beispiel:

SUBTTL Unterprogramm Kreis

Dieses Beispiel erzeugt den Untertitel "Unterprogramm Kreis".

SUBTTL

Hier wird als Untertitel eine leere Zeile generiert.

9.8. PAGE

Syntax:

```
PAGE <laenge>[,<breite>]
PAGE +
PAGE
```

Mittels PAGE kann die Seitenlaenge und die Zeilenbreite fuer die Assemblerliste festgelegt werden. Man kann den Abschnitt erhoehen und die Nummer des Abschnitts entsprechend anpassen oder einen Seitenwechsel erzwingen.

Wurden beide Parameter - <laenge> und <breite> - spezifiziert, wird die maximale Zeilenanzahl pro Seite auf <laenge> und die maximale Zeichenanzahl pro Zeile auf <breite> gesetzt. Die Zeilenanzahl kann sich in den Grenzen von 10 bis 255 und die Zeichenanzahl in den Grenzen von 60 bis 132 bewegen. Als Standard nimmt MASM fuer die Zeilenanzahl 50 und fuer die Zeichenanzahl 80 an. Wird nur die Breite, aber nicht die Laenge spezifiziert, muss vor Breite unbedingt ein Komma (,) geschrieben werden.

Das Zeichen Plus (+) nach PAGE bewirkt, dass die Abschnittsnummer um 1 erhoeht und die Seitennummer auf 1 zurueckgesetzt wird. Die Seitennummern der Assemblerliste haben folgende Form:

<abschnitt>-<seitennummer>

Dabei ist <abschnitt> die Abschnittsnummer innerhalb eines Moduls und <seitennummer> ist die Seitennummer innerhalb des Abschnitts. Standardmaessig wird mit 1-1 begonnen.

PAGE ohne Argumente bewirkt einen Vorschub auf die neue Seite und die Ausgabe der Titel- und Untertitelzeile.

Beispiele:

1. **PAGE**

Es wird ein Seitenvorschub erzeugt.

2. **PAGE 58,60**

Die maximale Zeilenanzahl wird auf 58, die maximale Zeichenanzahl auf 60 gesetzt.

3. **PAGE ,132**

Ab jetzt ist die maximale Zeichenanzahl 132. Die aktuelle Zeilenanzahl (entweder der Standard von 50 oder ein vorher gesetzter Wert) bleibt unveraendert erhalten.

4. **PAGE +**

Hier wird die Abschnittsnummer erhoeht und die Seitennummer auf

1 gesetzt. War beispielsweise die Seitennummer vorher 3-6, so erhaelt die naechste Seite die Nummer 4-1.

9.9. .LIST und .XLIST

Syntax:

```
.LIST
.XLIST
```

Mit den Pseudooperationen .LIST und .XLIST wird gesteuert, welche Quellzeilen in die Liste uebernommen werden. .XLIST unterdrueckt das Auflisten der nachfolgenden Quellzeilen. .LIST hebt die Wirkung von .XLIST wieder auf. Diese Pseudooperationen werden meistens paarweise verwendet, um zu verhindern, dass ein bestimmter Abschnitt einer Quelldatei in die Liste aufgenommen wird.

Die Pseudooperation .XLIST setzt alle anderen Listenpseudooperationen ausser Kraft (ausser .LIST).

Beispiel:

```
.XLIST          ;ab hier werden die Zeilen nicht in die
    .           ;Liste uebernommen
    .
    .
.LIST           ;die Zeilen werden ab jetzt wieder in
    .           ;die Liste uebernommen
    .
    .
```

9.10. .SFCOND, .LFCOND und .TFCOND

Syntax:

```
.SFCOND
.LFCOND
.TFCOND
```

Die Pseudooperationen .SFCOND und .LFCOND bestimmen, ob falsche bedingte Bloecke aufgelistet werden. .SFCOND unterdrueckt das Auflisten aller nachfolgenden falschen bedingten Bloecke. Die Anweisung .LFCOND laesst das Listen dieser Bloecke wieder zu. Aehnlich wie .LIST und .XLIST koennen diese Pseudooperationen benutzt werden, um das Auflisten von falschen bedingten Bloecken in Programmabschnitten zu unterdruecken.

.TFCOND wirkt in Verbindung mit dem /X-Schalter des Assemblers. Bei angegebenem /X-Schalter in der MASM-Befehlszeile setzt .TFCOND den Standard auf Listen der falschen bedingten Bloecke. Ist der /X-Schalter nicht angegeben, dann bewirkt .TFCOND das Unterdruecken der falschen bedingten Bloecke. Jedesmal, wenn ein

neues .TFCOND im Quellkode auftritt, wird das Listen der falschen bedingten Bloecke umgeschaltet. Es wird eingeschaltet, wenn es vorher aus war und umgekehrt.

.TFCOND ist unabhaengig von .LFCOND und .SFCOND.

Beispiel:

```
test1   DB      0       ;Symbol ist definiert --->
                        ;alle Bedingungen sind falsch

                        ;/X nicht verwendet  /X verwendet
.SFCOND
IFNDEF  test1           ;nicht gelistet      nicht gelistet
test2   DB      128
ENDIF

.LFCOND
IFNDEF  test1           ;aufgelistet         aufgelistet
test2   DB      128
ENDIF

.TFCOND
IFNDEF  test1           ;aufgelistet         nicht gelistet
test2   DB      128
ENDIF

.TFCOND
IFNDEF  test1           ;nicht gelistet      aufgelistet
test2   DB      128
ENDIF
```

Das Auflisten der letzten beiden Bedingungsbloecke kehrt sich um, wenn der /X-Schalter verwendet wird.

9.11. .LALL, .XALL und .SALL

Syntax:

```
.LALL
.XALL
.SALL
```

Die Pseudooperationen .LALL, .XALL und .SALL steuern das Auflisten der Anweisungen von Makros, die in die Quelldatei erweitert wurden. Die Makrodefinition wird immer vollstaendig aufgelistet, aber die Makroerweiterung nur bei entsprechend gesetzter Pseudooperation.

.LALL veranlasst den Assembler, alle Quellanweisungen in einem Makro aufzulisten, einschliesslich der Kommentare, denen ein einfaches Semikolon (;) vorausgeht. Ausgeschlossen werden die Kommentare, die mit einem doppelten Semikolon (;;) beginnen.

.XALL listet nur die Kode oder Daten generierenden Anweisungen.

Kommentare werden ignoriert.

.SALL unterdrueckt das Listen aller Makroerweiterungen. Das heisst, in der Quelliste erscheint nur die Makroaufrufzeile, nicht die Zeilen, die durch den Aufruf generiert werden.

Im nachfolgenden Beispiel wird angenommen, dass der folgende Makro am Anfang der Quelldatei definiert worden ist:

```
test    MACRO
;;Makrokommentarzeile
;normale Kommentarzeile
        IF2                     ;kein Kode oder Daten
        ASSUME  cs:code         ;kein Kode oder Daten
        DW      20 DUP(?)       ;generiert Daten
        mov     ax,bx           ;generiert Code
        ENDIF                   ;kein Kode oder Daten
        ENDM
```

Angenommen, dieser Makro wird in einer Quelldatei einmal mit jeder dieser Pseudooperationen aufgerufen:

```
.LALL
        test                    ;Aufruf mit .LALL
.XALL
        test                    ;Aufruf mit .XALL
.SALL
        test                    ;Aufruf mit .SALL
```

Beispiel:

```
                        .LALL
                             test
                      1      ;normale Kommentarzeile
                      1      IF2                ;Kein Kode oder
                                                ;Daten
                      1      ASSUME cs:code     ;Kein Kode oder
                                                ;Daten
0005  0014[                  DW      20 DUP(?) ;generiert Daten
            ????  1
                ] 1
002D  8B C3           1      mov     ax,bx     ;generiert Kode
                      1      ENDIF
                        .XALL
                             test
002F  0014[           1      DW      20 DUP(?) ;generiert Daten
0057  8B C3           1      mov     ax,bx     ;generiert Kode
                        .SALL
                             test
```

Es ist zu beachten, dass die Makrokommentarzeile in keiner Liste der Makroerweiterung erscheint. Normale Kommentare werden nur bei Verwendung von .LALL aufgelistet.

9.12. .CREF und .XCREF

Syntax:

```
.CREF
.XCREF [<name>[,<name>,...]]
```

Die Pseudooperationen .CREF und .XCREF steuern das Generieren der Crossreferenz fuer die Makroassembler-Crossreferenzdatei. .XCREF unterdrueckt das Generieren der Crossreferenz fuer Marken, Variable und Symbole. Wenn .CREF auftritt, wird das Generieren fortgesetzt.

Ist bei .XCREF <name> angegeben, dann wird die Crossreferenz nur fuer diese Marken, Variablen oder Symbole unterdrueckt. Alle anderen Namen werden beruecksichtigt. Die genannten Marken, Variablen oder Symbole werden in der Symboltabelle der Assemblerliste uebergangen. Wenn zwei oder mehr Namen angegeben sind, muessen sie durch Kommas (,) voneinander getrennt werden.

Beispiel:

```
.XCREF                  ;Unterdruecken der Crossreferenz
  .                     ;der Symbole in diesem Block
  .
  .
.CREF                   ;Symbole dieses Blockes wieder
  .                     ;in die Symboltabelle aufnehmen
  .
  .
.XCREF test1,test2      ;in diesem Block fuer test1 und
  .                     ;test2 keine Crossreferenz
  .                     ;erzeugen
  .
```

10. Abarbeiten des Makroassemblers

10.1. Einleitung

Der Makroassembler uebersetzt Assembler-Quelldateien und erzeugt Objektdateien, die unter dem Betriebssystem DCP gebunden und ausgefuehrt werden koennen.

Dieses Kapitel erklaert, wie MASM benutzt wird und beschreibt das Format der durch MASM erzeugten Assemblerliste.

10.2. Starten und Verwenden von MASM

Die Abschnitte 10.2.1. und 10.2.2. erklaeren, wie MASM zum Assemblieren einer Programmdatei gestartet und verwendet wird.

Dazu gibt es zwei verschiedene Methoden:

- durch Beantworten einer Reihe von Eingabeaufforderungen
- oder durch eine DCP-Befehlszeile.

Wenn MASM gestartet ist, werden entweder die spezifizierten Dateien oder die Eingabeaufforderungen fuer zusaetzliche Dateien abgearbeitet.

MASM kann jederzeit durch Druecken von <CTRL>+<C> abgebrochen werden.

10.2.1. Aufrufen des Assemblers mit Hilfe von Eingabeaufforderungen

Um den Assembler zu starten kann man direkt MASM eingeben. Folgende Schritte sind notwendig:

1. Eingeben

   MASM

   und druecken <ENTER> in der DCP-Befehlsebene. MASM zeigt darauf folgende Eingabeaufforderung an:

   Source filename [.ASM]:

2. Es ist der Name der zu uebersetzenden Datei einzugeben, einschliesslich Laufwerk und Pfad, wenn die Datei sich nicht im aktuellen Verzeichnis befindet. Anschliessend ist<ENTER> zu betaetigen. Wenn keine Dateierweiterung angegeben ist, ergaenzt der Assembler .ASM.

   Der Assembler verlangt einen Quelldateinamen. Man kann nicht nur <ENTER> eingeben, wie das bei anderen Eingabeaufforderungen der Fall ist.

   Als naechstes wird diese Eingabeaufforderung angezeigt:

   Object filename [source.OBJ]:

3. Es ist zu beachten, dass source der Name der spezifizierten Quelldatei ist. Es ist der Name fuer den Objektkode und <ENTER> einzugeben. War keine Dateierweiterung angegeben, nimmt der Assembler .OBJ als Standard. Wird der Standarddateiname gewuenscht (= Name der Quelldatei), ist kein Dateiname, sondern nur <ENTER> einzugeben.

   Als naechstes erscheint die folgende Eingabeaufforderung:

   Source listing [NUL.LST]:

4. Wenn der Assembler eine Liste erzeugen soll, muss hier der Name fuer die Listdatei und <ENTER> eingegeben werden. Falls keine Dateierweiterung angegeben wird, nimmt der

Assembler .LST als Standard an. Wird keine Assemblerliste gewuenscht, ist nur <ENTER> einzugeben.

Danach wird gefragt:

Cross-reference [NUL.CRF]:

5. Soll der Assembler eine Crossreferenzdatei erzeugen, ist der Name fuer diese Datei und <ENTER> einzugeben. Der Standard fuer die Dateierweiterung ist .CRF. Wird nur <ENTER> betaetigt, wird keine Crossreferenzdatei erzeugt.

Dann uebersetzt MASM die Quelldatei.

Am Ende jeder Eingabezeile koennen ein oder mehrere wahlfreie Schalter spezifiziert werden. Jedem Schalter muss ein Schraegstrich (/) oder ein Bindestrich (-) vorangehen. Die Schalter sind im Abschnitt 10.3. beschrieben.

Fuer jede sich nicht im aktuellen Verzeichnis befindende Datei muss der entsprechende Pfadname angegeben werden.

An jeder Eingabeaufforderung kann der Rest der Dateinamen im Befehlszeilenformat eingegeben werden. Zum Beispiel kann man fuer alle restlichen Eingabeaufforderungen die Standardantworten auswaehlen, in dem man ein Semikolon (;) nach der Eingabe oder Kommas (,) angibt, um die einzelnen Dateien kenntlich zu machen. Erkennt MASM ein Semikolon, arbeitet er die restlichen Eingabeaufforderungen ohne Anzeige ab und waehlt die Standardantworten aus.

Beispiele:

1. MASM

```
Source filename [.ASM]: prog
Object filename [.OBJ]: b:prog
Source listing [NUL.LST]: PRN /D
Cross-reference [NUL.CRF]: b:\cref\prog
```

Dieses Beispiel weist MASM an, die Quelldatei prog.asm vom aktuellen Laufwerk zu assemblieren und den Objektkode prog.obj auf das Laufwerk B in das aktuelle Verzeichnis zu schreiben. Der Geraetename und der Schalter /D nach der Eingabeaufforderung Source listing besagt, dass eine Assemblerliste (einschliesslich einer Pass 1 - Liste) auf den Drucker ausgegeben wird. Der /D-Schalter ist im Abschnitt 10.3.1. beschrieben. MASM gibt die Crossreferenzdaten in die Datei prog.crf in das Verzeichnis \cref auf das Laufwerk B aus.

2. MASM

```
Source filename [.ASM]: prog
Object filename [.OBJ]: f123;
```

In diesem Beispiel uebersetzt MASM die Quelldatei prog.asm und schreibt den Objektkode in die Datei fi23.obj. Das Semikolon nach dem Objektdateinamen besagt, dass fuer die restlichen Eingabeaufforderungen die Standarddateinamen auszuwaehlen sind. Das bedeutet, der Assembler erzeugt weder eine Assemblerliste noch eine Crossreferenz.

10.2.2. Aufrufen des Assemblers mit Befehlszeile

Man kann eine Programmquelldatei uebersetzen durch die Eingabe des MASM-Befehls und den Namen der Dateien, die zu verarbeiten sind. Die Befehlszeile hat folgende Form:

MASM <quelldatei>[,[<objektdatei>][,[<listdatei>]
[,[<crossreferenzdatei>]]]] [schalter] [;]

<quelldatei> muss der Name der zu assemblierenden Datei sein. Wurde keine Dateierweiterung angegeben, ergaenzt MASM .ASM. <schalter> kann eine Kombination von MASM-Schaltern sein (siehe Abschnitt 10.3.). Schalter koennen an beliebiger Stelle in der Befehlszeile stehen.

<objektdatei> ist wahlfrei. Es ist der Name, den die Objektkodedatei erhalten soll. Ist kein Name angegeben worden, verwendet MASM den Quelldateinamen mit der Dateierweiterung .OBJ.

<listdatei> ist ebenfalls wahlfrei und stellt den Namen der Datei fuer die Assemblerliste dar. War kein Name eingegeben, so wird diese Liste unterdrueckt. Die Assemblerliste enthaelt den uebersetzten Kode fuer jede Quellanweisung sowie die Namen und Typen der Symbole, die in dem Programm definiert sind. Wurde keine Dateierweiterung angegeben, traegt MASM .LST ein.

<crossreferenzdatei> ist der Name, den die Crossreferenzausgabe erhalten soll. Er kann ebenfalls weggelassen werden. Falls diese Datei erzeugt wurde, kann sie mit dem Programm CREF verarbeitet werden. Dieses Programm erzeugt eine Crossreferenzliste der Symbole des Programms fuer den Programmtest. Der Standard fuer die Dateierweiterung ist .CRF.

Man kann das Semikolon (;) in der Befehlszeile verwenden, um fuer die restlichen Dateinamen den Standard auszuwaehlen. Standard ist:

Uebersetzen Quelldatei
Ausgabe Objektdatei
Unterdruecken Assemblerliste
Unterdruecken Crossreferenz

Ein Semikolon nach dem Quelldateinamen bedeutet, dass der Standard-Objektkodename verwendet und weder Assemblerliste noch Crossreferenz erzeugt wird. Ein Semikolon nach dem Objektdateinamen unterdrueckt die Liste und die Crossreferenz. Ein Semikolon nach dem Listdateinamen unterdrueckt nur die Crossreferenz.

Alle waehrend der Uebersetzung erzeugten Dateien werden auf das aktuelle Laufwerk in das aktuelle Verzeichnis geschrieben, falls nichts anderes fuer die einzelnen Dateien spezifiziert ist. Es muss getrennt fuer jede Datei eine anderes Laufwerk bzw. ein Pfad spezifiziert werden, wenn diese Datei nicht in das aktuelle Verzeichnis gehen soll. Ebenso kann ein Geraetename an die Stelle des Dateinamens geschrieben werden, z. B.

NUL - keine Datei
PRN - Drucker.

Bemerkung:

Wird kein Semikolon verwendet, sind alle Kommas in der Befehlszeile erforderlich. Sind nicht alle Kommas geschrieben worden, dann werden die restlichen Eingaben wie oben beschrieben durch Eingabeaufforderungen abgefragt. Wird fuer einen Dateinamen der Standard gewuenscht, fuer einen nachfolgenden soll aber keine Eingabe erfolgen, so sind zwei aufeinanderfolgende Kommas zu schreiben (,,). Leerzeichen in einer Befehlszeile sind wahlfrei.

Ein fehlerhaft eingegebener Dateiname fuehrt zu einer Fehlermeldung, und MASM wiederholt die Eingabeaufforderung in der Art, wie sie im vorigen Abschnitt beschrieben ist.

Beispiele:

1. **MASM prog.asm, prog.obj, prog.lst, prog.crf**

Dieses Beispiel ist auquivalent zu:

2. **MASM prog,,,;**

Die Quelldatei prog.asm wird assembliert und in prog.obj der Objektcode geschrieben. Ausserdem wird eine Assemblerliste unter dem Namen prog.lst und eine Crossreferenzdatei prog.crf erzeugt.

3. **MASM startup,,stest;**

Die Quelldatei startup.asm wird assembliert und der Objektkode in die Datei mit dem Standardnamen startup.obj geschrieben. Ausserdem wird eine Assemblerliste mit dem Namen stest.lst erzeugt und eine Crossreferenzdatei durch die Angabe des Semikolons unterdrueckt.

4. **MASM B:\src\build;**

In diesem Beispiel wird MASM angewiesen, eine Quelldatei build.asm aus dem Verzeichnis \src auf dem Laufwerk B zu assemblieren. Das Semikolon bewirkt, dass eine Objektdatei mit dem Standardnamen build.obj in das aktuelle Verzeichnis geschrieben wird, aber es verhindert das Erstellen der Assemblerliste und der Crossreferenzdatei. Man beachte, dass die Objektdatei auf das aktuelle Laufwerk, nicht auf das fuer die Quelldatei spezi-

fizierte kommt.

10.3. Verwenden von MASM-Schaltern

Die MASM-Schalter steuern die Arbeit des Assemblers und das Format der Ausgabedateien wird generiert. MASM hat folgende Schalter:

Schalter	Bedeutung
/A	Segmente in alphabetischer Reihenfolge ordnen
/S	Segmente in der Quellcodereihenfolge schreiben
/B<zahl>	Puffergroesse setzen
/C	Crossreferenzdatei spezifizieren
/L	Assemblerliste spezifizieren
/D	Pass 1-Liste erzeugen
/D<symbol>	Assemblersymbol definieren
/I<pfad>	Suchpfad fuer Include-Dateien setzen
/ML	Gross- und Kleinschreibung bei Namen wird als verschieden betrachtet
/MX	Gross- und Kleinschreibung bei EXTRN- und PUBLIC-Namen wird als verschieden betrachtet
/MU	Namen in Grossbuchstaben wandeln
/N	Tabellen in der Listdatei unterdruecken
/P	auf unzulaessigen Kode pruefen
/R	Kode fuer Gleitkommaprozessor erzeugen
/E	Kode fuer Gleitkommaemulator erzeugen
/T	Meldungen bei erfolgreicher Uebersetzung unterdruecken
/V	eine Extrastatistik auf dem Bildschirm anzeigen
/X	Auch falsche Bedingungen in die Liste aufnehmen
/Z	Fehlerzeilen anzeigen

Die Schalter koennen irgendwo in der Befehlszeile des MASM stehen. Ein Schalter wirkt auf alle Dateien in der Befehlszeile, auch wenn er am Ende der Zeile eingetragen ist.

Schalter werden durch den Schraegstrich (/) oder den Bindestrich (-) und entweder durch kleine oder grosse Buchstaben gekennzeichnet. Beispielsweise sind folgende Schreibweisen aequivalent:

```
/a
/A
-a
-A
```

Mehrere Schalter sind durch Leerzeichen voneinander zu trennen.

Bemerkung:

Bindestriche sollten nicht in Quelldateinamen verwendet werden. Obwohl der Bindestrich fuer DCP-Dateinamen ein erlaubtes Zeichen ist, bewertet der Assembler diesen als Beginn eines Schalters. Zum Beispiel wird der Dateiname prog-c vom Assembler als Dateiname prog mit nachfolgendem Schalter -c interpretiert. Das Ergebnis ist eine Fehlermeldung.

10.3.1. Segmente in alphabetischer Reihenfolge schreiben

Syntax:

```
/A
```

Dieser Schalter weist die Anordnung der uebersetzten Segmente in alphabetischer Reihenfolge an. Wird dieser Schalter weggelassen, kopiert MASM die Segmente in der Reihenfolge in die Objektkodedatei, wie sie in der Quelldatei auftreten. Diese Information wird an den Programmverbinder uebergeben. Der Schalter hat keine Auswirkungen auf das Aussehen der Assemblerliste.

Beispiel:

```
MASM prog /A
```

Dieses Beispiel erzeugt eine Objektdatei, in der die Segmente in alphabetischer Reihenfolge angeordnet sind. Das heisst, wenn die Quelldatei prog.asm Segmente mit den Klassentypen 'DATA', 'CODE' und 'STACK' enthaelt, wird im Objektkode eine Information ueber folgende Segmentreihenfolge abgelegt: 'CODE', 'DATA', STACK'.

Auf die Bedeutung der Segmentreihenfolge und der Klassentypen wird im Einzelnen in der Beschreibung des Programmverbinders LINK und im Abschnitt 3.4.3. in dieser Dokumentation eingegangen.

10.3.2. Segmente in der Quellkodereihenfolge schreiben

Syntax:

/S

Der /S-Schalter veranlasst den Assembler, die Segmente in derselben Reihenfolge in die Objektdatei zu schreiben, wie sie in der Quelldatei erscheinen. Das ist die Standardreihenfolge.

10.3.3. Setzen Dateipuffergroesse

Syntax:

/B<zahl>

Der /B-Schalter weist den Assembler an, die Dateipuffergroesse fuer die Quelldatei zu aendern. <zahl> ist die Anzahl von 1024 Byte - (1k Byte -) Speicherbloecken, die als Puffer zugewiesen werden koennen. Die Puffergroesse kann in den Grenzen von 1 bis 63 k (nicht 64 k!) Byte ausgewaehlt werden. Die Standardpuffergroesse ist 32 k Byte.

Ein Puffer, groesser als die Quelldatei, ermoeglicht die vollstaendige Uebersetzung im Speicher und damit eine Vergroesserung der Uebersetzungsgeschwindigkeit.

Wenn fuer die gewuenschte Puffergroesse der Platz nicht ausreicht, weil der Speicher des Computers zu klein ist oder weil sich zu viele residente Programme im Speicher befinden, wird eine Fehlermeldung ueber ungenuegenden Speicherplatz angezeigt. Nach Reduzierung der Puffergroesse kann erneut versucht werden, das Programm zu uebersetzen.

Beispiele:

1. **prog,, /B16;**

Das Beispiel vermindert die Puffergroesse auf 16 k Byte.

2. **MASM,, /B63;**

Im Beispiel wird die Puffergroesse auf 63 k Byte vergroessert.

10.3.4. Erzeugen einer Pass 1 - Liste

Syntax:

/D

Der /D-Schalter bewirkt das Hinzufuegen einer Pass 1 - Liste an

die Assemblerliste, um das Ergebnis beider Assemblerpaesse zu sehen. Diese Pass 1 - Liste ist beim Suchen von Phasenfehlern nuetzlich. Phasenfehler entstehen, wenn der Assembler im Pass 1 Annahmen ueber das Programm trifft, die sich im Pass 2 nicht bestaetigen.

Pass 1 - Fehler werden wie normale Fehler in der Liste ausgegeben.

/D bewirkt nur dann eine Pass 1 - Liste, wenn auch eine Assemblerliste erstellt wird. Fehlermeldungen werden fuer beide Paesse angezeigt, auch wenn keine Assemblerliste erzeugt wird.

Weitere Informationen zur Pass 1 - Liste werden im Abschnitt 10.4.6. gegeben.

Beispiel:

```
MASM prog,, /D;
```

In diesem Beispiel wird eine Pass 1 - Liste fuer die Quelldatei prog.asm erzeugt und mit in der Datei prog.lst untergebracht.

10.3.5. Definieren Assemblersymbol

Syntax:

```
/D<symbol>
```

Mit diesem Schalter kann ein Symbol definiert werden, dass waehrend der Uebersetzung so verwendet werden kann, als waere es in der Quelldatei definiert worden.

Das spezifizierte Symbol wird als "Null-Text-Kette" definiert. Dieses so definierte Symbol kann durch die bedingten Assembler-Pseudooperationen IFDEF und IFNDEF ausgewertet werden. Diese Pseudooperationen sind im Abschnitt 7.2.3. erklaert.

Beispiel:

```
MASM prog,, /Dbreite
```

Das Beispiel definiert ein Symbol breite und gibt ihm den Wert 0. Das Symbol koennte in folgendem bedingten Assemblerblock verwendet werden

```
IFDEF breite
      PAGE 50,132
ENDIF
```

Ist das Symbol in der Befehlszeile definiert, wird die Assemblerliste fuer einen 132-Zeichen breiten Drucker formatiert. Ist das Symbol nicht definiert, wird die Standardbreite von 80

Zeichen genommen.

Siehe dazu auch die Beschreibung der Pseudooperation PAGE im Abschnitt 9.8.

10.3.6. Festlegen eines Suchpfades fuer INCLUDE-Dateien

Syntax:

/I<pfad>

Der /I-Schalter wird verwendet, um einen Suchpfad fuer INCLUDE-dateien festzulegen. Gesucht wird in der Reihenfolge der Pfad-angaben in der Befehlszeile.

Die Pseudooperation INCLUDE und INCLUDE-Dateien sind im Ab-schnitt 9.2. beschrieben.

Beispiel:

MASM prog,, /Ib:\io /I\makro;

Enthaelt die Quelldatei die Anweisung

INCLUDE progio.mac

sucht MASM die Datei progio.mac zuerst im Verzeichnis \io auf dem Laufwerk B, dann im Verzeichnis \makro des aktuellen Lauf-werks und zuletzt im aktuellen Verzeichnis.

Man sollte in der INCLUDE-Anweisung keinen Pfad spezifizieren, wenn man in der Befehlszeile einen Suchpfad angeben will. Ent-haelt eine Quelldatei folgende Anweisung

INCLUDE a:\makro \progio.mac;

so wird MASM die Datei progio.mac in a:\makro suchen und den in der Befehlszeile angegebenen Suchpfad ignorieren.

10.3.7. Gross-/Kleinschreibung bei Namen

Syntax:

/ML

Der /ML-Schalter erhaelt beim Assemblieren die Kleinschreibung in Marken, Variablen und Symbolnamen. Alle Namen, die die glei-che Rechtschreibung haben, aber verschiedene Buchstaben (grosse bzw. kleine) werden als verschieden betrachtet.

So sind zum Beispiel DATA und data bei Verwenden des Schalters /ML verschieden. Ohne diesen Schalter wurde der Assembler auto-matisch kleine Buchstaben in Grossbuchstaben wandeln.

Dieser Schalter ist zum Binden von Assemblermoduln mit Moduln, die durch einen die Gross-/Kleinschreibung beruecksichtigenden Compiler erzeugt wurden, notwendig.

Beispiel:

```
MASM prog /ML,,;
```

Dieses Beispiel weist den Assembler an, die Kleinschreibung in den in der Quelldatei definierten Namen zu erhalten.

10.3.8. Gross-/Kleinschreibung bei Eintrittspunkten und externen Namen

Syntax:

```
/MX
```

Der /MX-Schalter erhaelt beim Assemblieren die Kleinschreibung in Eintrittspunkten und externen Namen. Alle anderen Namen werden durch MASM in Grossbuchstaben gewandelt. Eintrittspunkte und externe Namen sind Marken, Variable oder Symbole, die mit den Pseudooperationen PUBLIC und EXTRN definiert wurden. Wurde der /MX-Schalter spezifiziert, schreibt der Assembler die Eintrittspunkte und externen Namen exakt so in die Objektdatei, wie sie in der Quelldatei auftreten.

Die Namen DATA und Data wuerden bei Verwenden des /MX-Schalters in der Objektdatei verschieden geschrieben.

Dieser Schalter wird verwendet, wenn Assemblermoduln mit durch Compiler erzeugte Moduln gebunden werden sollen, die diese Eigenschaft besitzen.

Beispiel:

```
MASM prog /MX,,,;
```

Dieses Beispiel veranlasst MASM die Kleinschreibung in jedem Eintrittspunkt und jedem externen Namen beizubehalten.

10.3.9. Namen in Grossbuchstaben wandeln

Syntax:

```
/MU
```

Dieser Schalter bewirkt, dass in Eintrittspunkten und externen Namen Kleinbuchstaben in Grossbuchstaben gewandelt werden. Das ist Standard.

10.3.10. Unterdruecken der Tabellen in der Assemblerliste

Syntax:

```
/N
```

Dieser Schalter teilt dem Assembler mit, dass am Ende der Assemblerliste alle Tabellen wegzulassen sind. Wenn dieser Schalter nicht ausgewaehlt wurde, enthaelt die Assemblerliste Tabellen ueber Makros, Strukturen und Records, Segmente, Gruppen und Symbole. Der Kodeteil der Liste wird durch den Schalter /N ebensowenig veraendert wie die Endemeldung.

Beispiel:

```
MASM prog,, /N;
```

10.3.11. Pruefen auf unzulaessigen Kode

Syntax:

```
/P
```

Der /P-Schalter bewirkt das Pruefen auf unzulaessigen Kode im geschuetzten Modus des 80286. Dieser Schalter ist nur dann wirksam, wenn der Assembler durch die Pseudooperation .286p gesteuert wird.

Die Pseudooperation .286p und andere zur Auswahl der Anweisungsliste sind im Abschnitt 3.3. erklaert.

Kode, der Daten mittels Segmentpraefix CS: in den Speicher transportiert, wird im nichtgeschuetzten 286-Modus und im 8086- und 80186-Modus akzeptiert. Er kann die Ursache fuer Probleme im geschuetzten Modus sein. Wirkt der /P-Schalter und tritt solcher Kode auf, wird Fehler 100 generiert.

Beispiel:

```
MASM prog /P;
```

Dieses Beispiel veranlasst den Assembler zu pruefen, ob Anweisungen Daten mit dem Segmentpraefix CS: unmittelbar in den Speicher transportieren.

10.3.12. Kode fuer einen Gleitkommaprozessor erzeugen

Syntax:

```
/R
```

Dieser Schalter weist den Assembler an, Kode fuer Gleitkomma-Anweisungen zu generieren, der durch die Prozessoren 8087 oder 80287 ausgefuehrt werden kann. Programme, die mit dem /R-Schalter erzeugt wurden, koennen nur auf Anlagen abgearbeitet werden, die entweder mit dem Arithmetikprozessor 8087 oder 80287 bestueckt sind.

10.3.13. Kode fuer einen Gleitkommaemulator erzeugen

Syntax:

```
/E
```

Mit dem Schalter /E ist es moeglich, Anweisungskode zu erzeugen, der den Gleitkommaprozessor 8087 oder 80287 emuliert. Dazu ist eine mathematische Emulationsbibliothek erforderlich, wie sie fuer die Sprachen C, PASCAL oder FORTRAN existiert.

Der Makroassembler besitzt **keine** mathematische Emulationsbibliothek.

Sollen Programme mit 8087- oder 80287-Befehlen abgearbeitet werden und der entsprechende Gleitkommaprozessor steht nicht zur Verfuegung, ist das Programm mit dem /E-Schalter zu uebersetzen und anschliessend mit einer mathematischen Emulationsbibliothek zu verbinden. Diese Bibliothek muss die die Gleitkommaanweisungen des 8087 bzw. 80287 emulierenden Routinen enthalten.

Beispiel:

```
MASM prog /E;
LINK prog,,,math.lib
```

In diesem Beispiel wird fuer die Gleitkommaanweisungen Emulationscode erzeugt. In der zweiten Befehlszeile wird die Objektdatei mit einer mathematischen Bibliothek verbunden.

Versucht man, den /E-Schalter ohne mathematische Bibliothek zu verwenden, wird die Uebersetzung fehlerfrei sein. Jedoch beim Binden der Objektdatei treten Fehler auf.

10.3.14. Zusaetzliche Assemblerstatistik anzeigen

Syntax:

```
/V
```

Der /V-Schalter bewirkt, dass am Ende eines Assemblerlaufes zusaetzlich zu den Angaben ueber die Anzahl von Fehlern und Warnungen sowie ueber den freien Bereich weitere Angaben auf den Bildschirm ausgegeben werden.

Diese sind:

Anzahl der Quellzeilen
Anzahl der Zeilen total
Anzahl der Symbole

10.3.15. Falsche Bedingungen listen

Syntax:

/X

Der /X-Schalter veranlasst den Assembler, alle Anweisungen, die den Koerper einer bedingten Pseudooperation bilden und deren Bedingung falsch ist, in die Assemblerliste auszugeben. Ist der /X-Schalter nicht in der Befehlszeile angegeben, werden diese Anweisungen in der Liste unterdrueckt. Dieser Schalter bietet die Moeglichkeit, auch die Anweisungen zu sehen, die keinen Kode erzeugen.

Der /X-Schalter laesst sich fuer alle bedingten Pseudooperationen anwenden: IF, IFE, IF1, IF2, IFDEF, IFNDEF, IFB, IFNB, IFIDN, IFDIF. Diese bedingten Pseudooperationen sind im Abschnitt 7.2. beschrieben.

Durch die Pseudooperation .TFCOND wird die Wirkung des /X-Schalters veraendert. Ein .SFCOND in der Quelldatei unterdrueckt das Listen falscher bedingter Bloecke waehrend ein .LFCOND das Auflisten falscher bedingter Bloecke wieder einschaltet. Diese beiden Pseudooperationen wirken ohne Ruecksicht darauf, ob der /X-Schalter in der Befehlszeile gesetzt war oder nicht.

Ein .TFCOND in der Quelldatei kehrt die normale Bedeutung des /X-Schalters um. Wenn der /X-Schalter gesetzt war und der Assembler auf ein .TFCOND in der Quelldatei trifft, werden nachfolgend falsche bedingte Bloecke unterdrueckt. Das naechste .TFCOND bewirkt wieder das Auflisten der falschen bedingten Bloecke.

Im folgenden wird die Wirkung von .TFCOND, .SFCOND, .LFCOND und dem /X-Schalter erklaert:

.SFCOND Der /X-Schalter hat keine Wirkung, falsche bedingte Bloecke werden nicht aufgelistet

.LFCOND Der /X-Schalter hat keine Wirkung, falsche bedingte Bloecke werden aufgelistet

.TFCOND Wechseln zwischen Unterdruecken und Listen falscher bedingter Bloecke

keine Pseuoperation
Listen der falschen bedingten Bloecke

Der /X-Schalter ist wirkungslos, wenn keine Assemblerliste

erzeugt wird. Weitere Informationen ueber die Pseudooperationen, die das Listen falscher bedingter Bloecke steuern, sind im Abschnitt 9.10. gegeben.

Beispiel:

```
MASM prog,,/X;
```

Tritt in der Quelldatei zweimal .TFCOND auf, beginnt der Assembler die falschen Bedingungen aufzulisten bis zum ersten Auftreten der Pseudooperation und setzt das fort nach dem Erreichen der zweiten. Das Wechseln zwischen Auflisten und Unterdruecken wird so mit jedem neuen Auftreten eines .TFCOND fortgesetzet.

10.3.16. Fehlerzeilen auf dem Bildschirm anzeigen

Syntax:

```
/Z
```

Der /Z-Schalter bewirkt, dass Zeilen, die einen Fehler enthalten auf dem Bildschirm angezeigt werden. Normalerweise wird nur eine den Fehler beschreibende Meldung ausgegeben. Mit dem /Z-Schalter wird sowohl die fehlerhafte Zeile als auch die Fehlermeldung angezeigt.

MASM assembliert ohne den /Z-Schalter schneller.

10.3.17. Crossreferenzdatei spezifizieren

Syntax:

```
/C
```

Mit dem /C-Schalter kann erreicht werden, dass eine Crossreferenzdatei erzeugt wird, auch wenn dies nicht in der Befehlszeile oder als Antwort auf die entsprechende Eingabeaufforderung spezifiziert wurde. Die Crossreferenzdatei hat dann den Namen der Quelldatei und die Erweiterung .CRF. Mit diesem Schalter kann kein Name fuer diese Datei vereinbart werden.

10.3.18. Assemblerliste spezifizieren

Syntax:

```
/L
```

Durch diesen Schalter kann eine Assemblerliste erzeugt werden, auch wenn das nicht in der Befehlszeile spezifiziert, oder als Antwort auf die entsprechende Eingabeaufforderung gegeben wurde. Die durch den /L-Schalter erzeugte Assmblerliste hat immer den Namen der Quelldatei und die Erweiterung .LST. Man kann mit

diesem Schalter keinen Namen vereinbaren.

10.3.19. Meldungen bei erfolgreichem Assemblieren unterdruecken

Syntax:

/T

Durch den /T-Schalter werden alle Meldungen unterdrueckt, wenn die Quelldatei ohne Fehler und Warnungen uebersetzt wurde.

Dieser Schalter kann sinnvoll in Stapelverarbeitungsdateien zum Unterdruecken unnoetiger Meldungen auf den Bildschirm verwendet werden.

10.4. Hinweise zum Lesen der Assemblerliste

MASM erzeugt eine Assemblerliste der Quelldatei nur dann, wenn dafuer ein Name in der Befehlszeile bzw. eine Antwort auf die entsprechende Eingabeaufforderung gegeben wurde oder wenn der /L-Schalter gesetzt war. In der Assemblerliste sind die Quellzeilen und der fuer jede Anweisung generierte Kode enthalten. Die Liste zeigt ausserdem alle Namen und Werte der Marken, Variablen und Symbole dieser Quelldatei.

Der Assembler erzeugt Tabellen fuer Makros, Strukturen, Records, Segmente, Gruppen und andere Symbole. Diese Tabellen stehen am Ende der Assemblerliste, es sei denn, sie wurden mit dem /N-Schalter unterdrueckt.

MASM listet nur den Typ der im Programm vorkommenden Symbole auf. Enthaelt das Programm keine Makros, wird kein Makroabschnitt in der Tabelle ausgegeben.

Ausserdem enthaelt die Assemblerliste Fehlermeldungen, falls waehrend der Uebersetzung Fehler aufgetreten sind. Diese Fehlermeldung wird unter der verursachenden Anweisungszeile ausgegeben. Am Ende der Liste werden die Anzahl Fehler und die Anzahl Warnungen mitgeteilt.

10.4.1. Hinweise zum Lesen des Kodes der Assemblerliste

Der Assembler listet den fuer die Anweisungen der Quelldatei generierten Maschinenkode auf.

Jede Zeile hat die Form:

[<zeilennummer>] <offset> <kode> <anweisung>

<zeilennummer> ist wahlfrei. Sie stellt die Nummer der Zeile beginnend bei der ersten Anweisung in der Assemblerliste dar. Diese Zeilennummer wird nur erzeugt, wenn eine Crossreferenz-

datei gefordert wird. Die Zeilennummer in der Liste entspricht nicht immer der Zeilennummer derselben Zeile in der Quelldatei.

<offset> ist die relative Adresse zum Anfang des Kodesegmentes.

<kode> ist der aktuelle Befehls- oder Datenkode, der fuer die Anweisung generiert wurde. Wenn moeglich, gibt MASM den aktuellen numerischen Wert fuer den Kode hexadezimal an. Sonst wird er kommentiert, wenn erst beim Binden der exakte Kode erzeugt werden kann:

```
---- R fuer einen Segmentoffset
---- E fuer eine externe Bezugnahme.
```

<anweisung> ist die Quellanweisung, die exakt so gezeigt wird, wie sie in der Quelldatei steht oder sie ist aus einem Makro erweitert worden.

Tritt waehrend der Uebersetzung ein Fehler auf, so werden unmittelbar unter der verursachenden Anweisung die Fehlernummer und die Fehlermeldung ausgegeben. Im Anhang A sind alle Fehler von MASM aufgefuehrt.

Eine Fehlermeldung enthaelt folgende Informationen:

- Quelldateiname
- Quellzeilennummer
- Fehlernummer
- Fehlermeldung

Beispiel:

```
        28                                  nov     ds,ax
Test2.ASM(22) : error 10 : Syntax
```

Die 22 in der Fehlermeldung ist die Zeilennummer der Quelldatei. Die 28 in der Kodezeile ist die Zeilennummer der Assemblerliste. Diese Zeilennummern muessen nicht identisch sein. Zeilennummern in der Assemblerliste werden nur erzeugt, wenn eine Crossreferenzdatei aufgebaut werden soll.

Der Assembler verwendet spezielle Symbole zum Kennzeichnen der Adressen, die von LINK aufgeloest werden muessen, oder zum Kennzeichnen von Werten, die auf einem speziellen Weg generiert werden. Diese Symbole sind in der folgenden Tabelle erklaert:

Symbol	Bedeutung
R	Verschiebliche Adresse; muss von LINK aufgeloest werden
E	Externe Bezugnahme; muss von LINK aufgeloest werden
----	Segment- oder Gruppenadresse; muss von LINK aufgeloest werden
=	Durch Pseudooperation EQU oder = definiert
nn:	Segmentpraefix in der Anweisung (nn - Hexakode des Segmentpraefix)
nn/	Praefix fuer REP- oder LOCK-Anweisung (nn - Hexakode fuer REP, REPZ, REPNZ, LOCK)
nn[xx]	DUP-Ausdruck (nn-mal den Wert xx kopieren)
n	Schachtelungstiefe bei Makroerweiterungen (n = 1...9 bzw. n = +, wenn groesser als 9)
C	Zeile stammt aus einer INCLUDE-Datei

Beispiel:

Im folgenden wird ein umfangreiches Beispiel gezeigt.

Es wird eine Datei, deren Name ueber die Tastatur eingeben wird, sequentiell gelesen. Die Woerter werden gezaehlt. Das Ergebnis erscheint auf dem Bildschirm hexadezimal.

Die Angabe LENGTH = in der Symboltabelle musste auf Grund der geringen Zeilenlaenge auf eine neue Zeile geschrieben werden. Normalerweise gehoeren diese Angaben an das Ende der vorhergehenden Zeile. Aus dem gleichen Grund wurden Kommentare auf extra Zeilen geschrieben. Das ist beim Assembler nicht Bedingung.

```
MASM (DCPX) V1.0                                1/4/87 00:09:48

TEST1.ASM                                         Page     1-1


                                        page    56,60
                                        TITLE   TEST1.ASM
                                dcpint  MACRO   funktion

                                ;Aufruf DCP-Funktion
                                        mov     ah,funktion

                                ;Uebernahme Funktionswert
                                        int     21H
                                        ENDM

                                error   MACRO   errnum
                                ;Anseige ERROR und EXIT
                                        mov     dx,OFFSET err&errnum
                                ;AAD Fehlertext
                                        dcpint  09H
                                ;Fkt. Anzeige Zeichenkette

                                        mov     al,errnum
                                ;EXIT mit Rueckgabe Fehlernummer
                                        dcpint  4CH
                                ;Fkt. Programmende
                                        ENDM

                                PUBLIC  eing,namepuff,dat_name,puffer
                                PUBLIC  err1,err2,zaehler,neu_merk
                                PUBLIC  eing_datei,open_datei,weiter
                                PUBLIC  puffer_lesen,abschluss
                                PUBLIC  conv_hex,rotieren
                                PUBLIC  quit,wort_zaehl,folge_zeich
                                PUBLIC  neu_wort,alt_wort,ausg_wort
                                PUBLIC  eing_ausg

0000                            stack   SEGMENT word stack 'STACK'
0000  0100[                            db      100H DUP(?)
          ??
                          ]

                                ;Stackbereich 256 Byte
0100                            stack   ENDS

0000                            data    SEGMENT word public 'DATA'
0000  45 69 6E 67 61 62 65      eing     db     'Eingabe Dateiname:$'
      20 44 61 74 65 69 6E
      61 6D 65 3A 20 24
0014  15 ??                     namepuff db     15H,?
                                ;Max. Laenge Dateiname
0016  0015[                     dat_name db     15H DUP(?)
          ??
                          ]
                                ;= 21 Byte Dateiname
```

```
002B  0400[                     puffer   dw      400H DUP(?)
          ????
                          ]

                                ;Puffergroesse = 2048 Byte

                                err1
082B  44 61 74 65 69 20 6E      db       'Datei nicht gefunden',0DH,0AH,'$'
      69 63 68 74 20 67 65
      66 75 6E 64 65 6E 0D
      0A 24
0842  45 2F 41 20 46 65 68      err2     db      'E/A Fehler',0DH,0A
      6C 65 72 0D 0A 24
084F  0000                      zaehler  dw      0
                                ;Wortzaehler = 0
0851  01                        neu_merk db      1
                                ;Neues Wortkennzeichen = 1

0852                            data     ENDS

0000                            code     SEGMENT byte public 'CODE'
                                         ASSUME  cs:code,ds:data

0000  B8 ---- R                 start:   mov     ax,data
                                ;Einstellen DS-Register
0003  8E D8                              mov     ds,ax
0005  BA 0000 R                          mov     dx,OFFSET eing
                                ;AAD Text
                                         dcpint  09H
0008  B4 09                   1          mov     ah,09H
000A  CD 21                   1          int     21H
                                ;Textanzeige
000C                            eing_datei:
000C  BA 0014 R                          mov     dx,OFFSET namepuff
                                ;Ladeadresse fuer Dateiname
                                         dcpint  0AH
000F  B4 0A                   1          mov     ah,0AH
0011  CD 21                   1          int     21H
                                ;Eingabe Dateiname
0013  8B F2                              mov     si,dx
                                ;SI einstellen auf Anfang Dateiname
0015  8A 5C 01                           mov     bl,BYTE PTR [si+1]
                                ;Bereitstellen 1 Byte in B
0018  C6 40 02 00                        mov     BYTE PTR [si+bx+2],0
                                ;Loeschen Zeilenende
001C  B2 0A                              mov     dl,0AH
                                ;Zeilenschaltung
                                         dcpint  02H
001E  B4 02                   1          mov     ah,02H
0020  CD 21                   1          int     21H
                                ;Druckausgabe
```

```
MASM (DCPX) V1.0                                1/4/87 00:09:48

TEST1.ASM                                         Page     1-3


0022                            open_datei:
0022  BA 0016 R                          mov     dx,OFFSET dat_name
                                ;AAD Dateiname
0025  32 C0                              xor     al,al
                                ;Wert 0 fuer Eroeffnen zum Lesen
                                         dcpint  3DH
0027  B4 3D                   1          mov     ah,3DH
0029  CD 21                   1          int     21H
                                ;Dateieroeffnung
002B  73 0D                              jnc     weiter
                                ;zur Weiterarbeit
002D                            fehler: error   1
002D  BA 082B R                1          mov     dx,OFFSET err1
0030  B4 09                    2          mov     ah,09H
0032  CD 21                    2          int     21H
0034  B0 01                    1          mov     al,1
0036  B4 4C                    2          mov     ah,4CH
0038  CD 21                    2          int     21H
                                ;Fehlermakro Fehler 1

003A  8B D8                     weiter: mov     bx,ax
                                ;Dateinummer nach BX
                                         mov     dx,OFFSET puffer
test1.ASM(95) : error 10: Syntax error
                                ;AAD Einlesebereich
003C                            ea_wiederh:
003C  B9 0800                            mov     cx,800H
                                ;Puffergroesse
                                         dcpint  3FH
003F  B4 3F                    1          mov     ah,3FH
0041  CD 21                    1          int     21H
                                ;Einlesen Datei 1 Block
0043                            puffer_lesen:
0043  72 0A                              jc      ea_err
                                ;Lesefehler
0045  3D 0000                            cmp     ax,0
                                ;Dateiende ?
0048  74 12                              je      abschluss
                                ;zum Dateischliessen
004A  E8 0088 R                          call    wort_zaehl
                                ;zur Anzeige Text
004D  EB ED                              jmp     SHORT ea_wiederh
                                ;naechster Satz
004F                            ea_err: error   2
004F  BA 0842 R                1          mov     dx,OFFSET err2
0052  B4 09                    2          mov     ah,09H
0054  CD 21                    2          int     21H
0056  B0 02                    1          mov     al,2
0058  B4 4C                    2          mov     ah,4CH
005A  CD 21                    2          int     21H
                                ;Fehler 2
```

```
005C                            abschluss:      dcpint  3EH
005C  B4 3E                   1         mov     ah,3EH
005E  CD 21                   1         int     21H
                                ;Dateiabschluss (Dateinr. in BX)

0060  8B 1E 084F R                      mov     bx,zaehler
                                ;Einstellen Wortzaehler
0064                            conv_hex:
0064  B1 04                             mov     cl,4
                                ;Bitanzahl zum Rotieren
0066  B5 04                             mov     ch,4
                                ;Anzahl Zeichen
0068                            rotieren:
0068  D3 C3                             rol     bx,cl
                                ;Rotieren linkes Zeichen nach rechts
006A  8A D3                             mov     dl,bl
                                ;einstellen DL fuer Wandlung

006C  80 E2 0F                          and     dl,0FH
                                ;Maske fuer linkes Zeichen

006F  80 C2 30                          add     dl,30H
                                ;konvertieren in ASCII-Zeichen
0072  80 FE 3A                          cmp     dh,3AH
                                ;Test auf groesser 9?
0075  7C 03                             jl      ausg
                                ;wenn nicht, dann Anzeige
0077  80 C2 07                          add     dl,07H
                                ;Erzeugen HEX-Buchstabe
007A                            ausg:   dcpint  02H
007A  B4 02                   1         mov     ah,02H
007C  CD 21                   1         int     21H
                                ;Zeichenanzeigefunktion
007E  FE CD                             dec     ch
                                ;Zeichenzaehler - 1
0080  75 E6                             jnz     rotieren
                                ;zum naechsten Zeichen

0082  32 C0                     quit:   xor     al,al
                                ;0 setzen fuer Rueckkehrcode
                                        dcpint  4CH
0084  B4 4C                   1         mov     ah,4CH
0086  CD 21                   1         int     21H
                                ;Programmbeendigung

0088                            wort_zaehl PROC NEAR
                                ;Prozedur Wortaufbereitung im Puffer
0088  53                               push    bx
                                ;retten Dateinummer
0089  BE 002A R                        mov     si,OFFSET puffer-1
                                ;ADR 1 Byte vor dem Puffer
```

```
MASM (DCPX) V1.0                                1/4/87 00:09:48

TEST1.ASM                                         Page     1-5


 008C  BB 0000                          mov     bx,0
                                      ;BX = 0 fuer Wortzaehler
 008F  8B C8                            mov     cx,ax
                                      ;Uebergabe 1 gelesenes Zeichen in CX
 0091  8A 26 0851 R                     mov     ah,neu_merk
                                      ;Bereitstellen Merkmal
 0095                                 folge_zeich:
 0095  46                               inc     si
                                      ;Index + 1
 0096  8A 04                            mov     al,[si]
                                      ;naechstes Zeichen
 0098  3C 20                            cmp     al,20H
                                      ;Test auf Space
 009A  7E 10                            jle     ausg_wort
                                      ;zur Wortausgabe
 009C  80 FC 01                         cmp     ah,1
                                      ;Test Wortmerkmal = 1
 009F  74 03                            je      neu_wort
                                      ;zum neuen Wort
 00A1  EB 04 90                         jmp     alt_wort
                                      ;zum naechsten Zeichen des Wortes

 00A4                                 neu_wort:
 00A4  43                               inc     bx
                                      ;Wortzaehler +1
 00A5  32 E4                            xor     ah,ah
                                      ;Loeschen Wortmerkmal
 00A7                                 alt_wort:
 00A7  E2 EC                            loop    folge_zeich
                                      ;gehe zum naechsten Zeichen
 00A9  EB 05 90                         jmp     eing_ausg
                                      ;gehe zum zaehlen
 00AC                                 ausg_wort:
 00AC  B4 01                            mov     ah,1
                                      ;Setzen Wortmerkmal auf 1
 00AE  E2 E5                            loop    folge_zeich
                                      ;gehe zum naechsten Zeichen

 00B0                                 eing_ausg:
 00B0  01 1E 084F R                     add     zaehler,bx
                                      ;Weiterzaehlen Wortzaehler
 00B4  88 26 0851 R                     mov     neu_merk,ah
                                      ;Speichern Wortmerkmal neu
 00B8  5B                               pop     bx
                                      ;Zurueckholen Dateinummer
 00B9  C3                               ret
                                      wort_zaehl ENDP

 00BA                                 code    ENDS

                                              END     start
```

Macros:

Name	Lines
DCPINT	4
ERROR	9

Segments and Groups:

Name	Size	Align	Combine	Class
CODE	00BA	BYTE	PUBLIC	'CODE'
DATA	0852	WORD	PUBLIC	'DATA'
STACK	0100	WORD	STACK	'STACK'

Symbols:

Name	Type	Value	Attr	
ABSCHLUSS	L NEAR	005C	CODE	Global
ALT_WORT	L NEAR	00A7	CODE	Global
AUSG	L NEAR	007A	CODE	
AUSG_WORT	L NEAR	00AC	CODE	Global
CONV_HEX	L NEAR	0064	CODE	Global
DAT_NAME	L BYTE	0016	DATA	Global
			Length = 0015	
EA_ERR	L NEAR	004F	CODE	
EA_WIEDERH	L NEAR	003C	CODE	
EING	L BYTE	0000	DATA	Global
EING_AUSG	L NEAR	00B0	CODE	Global
EING_DATEI	L NEAR	000C	CODE	Global
ERR1	L BYTE	082B	DATA	Global
ERR2	L BYTE	0842	DATA	Global
FEHLER	L NEAR	002D	CODE	
FOLGE_ZEICH	L NEAR	0095	CODE	Global
NAMEPUFF	L BYTE	0014	DATA	Global
NEU_MERK	L BYTE	0851	DATA	Global
NEU_WORT	L NEAR	00A4	CODE	Global
rOPEN_DATEI	L NEAR	0022	CODE	Global
PUFFER	L WORD	002B	DATA	Global
			Length = 0400	
PUFFER_LESEN	L NEAR	0043	CODE	Global
QUIT	L NEAR	0082	CODE	Global

```
                    *** MASM ***

MASM (DCPX) V1.0                                1/4/87 00:09:48

TEST1.ASM                                              Symbols-2


ROTIEREN . . . . . .          L NEAR  0068    CODE    Global

START  . . . . . .            L NEAR  0000    CODE

WEITER . . . . . .            L NEAR  003A    CODE    Global
WORT_ZAEHL . . . . . .        N PROC  0088    CODE    Global
                                                       Length = 0032

ZAEHLER  . . . . . .          L WORD  084F    DATA    Global


    209 Source  Lines
    275 Total   Lines
     53 Symbols

  49094 Bytes symbol space free

      0 Warning Errors
      1 Severe  Errors
```

10.4.2. Hinweise zum Lesen einer Makrotabelle

Die Tabelle am Ende der Assemblerliste zeigt die Namen und die Gesamtzeilenzahl aller in der Quelldatei definierten Makros. Die Liste hat zwei Spalten mit den Ueberschriften

N a m e Lines

Beispiel:

N a m e	Lines
DCPINT	4
ERROR	9

Die Namensspalte enthaelt alle Makronamen in alphabetischer Reihenfolge. Diese sind exakt so geschrieben, wie in der Quelldatei angegeben, mit der Ausnahme, dass kleine Buchstaben in grosse gewandelt werden. Das kann mittels /ML-Schalter verhindert werden. Namen, die laenger als 31 Zeichen sind, werden abgeschnitten. Die Spalte Lines zeigt die Zeilenanzahl des Makros an.

10.4.3. Hinweise zum Lesen einer Struktur- und Recordtabelle

Die Tabelle am Ende der Liste enthaelt die Namen und Dimensionen aller Strukturen und Records in der Quelldatei.

Die Namensspalte enthaelt den Namen der Struktur oder des Records. Darauf folgen eingerueckt Zeilen mit den Namen der Felder im Inneren der Strukturen oder Records.

Die Namen sind exakt so geschrieben, wie in der Quelldatei, mit der Ausnahme, dass kleine Buchstaben in grosse gewandelt werden, wenn das nicht mit dem /ML-Schalter verhindert wird. Namen, die laenger als 31 Zeichen sind, werden abgeschnitten.

Beispiel:

N a m e	Width Shift	# fields Width	Mask	Initial
STRUC1	001A	0003		
COUNT	0000			
VALUE	0001			
NAME	0015			

Fuer eine Struktur steht in der Spalte Width die Groesse des Records in Bytes. Unter #fields steht die Anzahl der Felder der Struktur. Beide Werte sind hexadezimal.

Fuer einen Record steht in der Width-Spalte die Groesse des Records in Bits. Die Spalte #fields enthaelt die Anzahl Felder

des Records.

Fuer die Felder von Strukturen bedeutet der Wert in der Spalte Shift die relative Adresse in Bytes vom Beginn der Struktur aus. Diese Angabe ist hexadezimal. Die anderen Spalten werden nicht benoetigt.

Beispiel:

```
                N a m e          Width    # fields
                                 Shift    Width   Mask   Initial

RECO . . . . . .                 000B     0002
  FL1  . . . . . .               0003     0008    07F8   0400
  FL2  . . . . . .               0000     0003    0007   0002
REC1 . . . . . .                 000A     0003
  FL1  . . . . . .               0006     0004    03C0   0000
  FL2  . . . . . .               0003     0003    0038   0000
  FL3  . . . . . .               0000     0003    0007   0000
```

Fuer die Felder eines Records stellt der Wert in der Spalte Shift den Abstand vom niederwertigsten Bit des Records zum niederwertigsten Bit des Feldes dar. Die Spalte Width enthaelt die Feldlaenge in Bits. Der Wert in der Spalte Mask stellt hexadezimal den Maximalwert dar, den das Feld annehmen kann. Falls ein Initialisierungswert angegeben war, steht dieser in der Spalte Initial. Die Maske und der Initialisierungswert sind so dargestellt, wie sie im Record plaziert sind und als ob alle anderen Felder Null waeren.

10.4.4. Hinweise zum Lesen einer Segment- und Gruppentabelle

Das folgende Beispiel zeigt den Teil der Assemblerliste, der die Namen, Groessen und Attribute aller Segmente und Gruppen der Quelldatei enthaelt.

Beispiel:

```
                N a m e          Size   Align   combine Class

DGROUP . . . . . .               GROUP
  DATA . . . . . .               0024   WORD    PUBLIC  'DATA'
  STACK  . . . . . .             0014   WORD    STACK   'STACK'
  CONST  . . . . . .             0000   WORD    PUBLIC  'CONST'
  HEAP . . . . . .               0000   WORD    PUBLIC  'MEMORY'
  MEMORY . . . . . .             0000   WORD    PUBLIC  'MEMORY'
FIRST  . . . . . .               0037   WORD    PUBLIC  'CODE'
MAIN_STARTUP . . . . . .         007F   PARA    NONE
```

Diese Tabelle hat fuenf Spalten:

```
N a m e          Size   Align   combine Class
```

Die Namensspalte enthaelt die Namen aller Segmente und Gruppen in alphabetischer Reihenfolge, mit Ausnahme der zu einer Gruppe gehoerenden Segmentnamen. Diese werden unter dem Gruppennamen angeordnet. Die Schreibweise der Namen ist gleich der in der Quelldatei, wobei kleine Buchstaben in grosse gewandelt werden, es sei denn, dass das mit dem Schalter /ML verhindert wird. Namen, die laenger als 31 Zeichen sind, werden abgeschnitten.

In der Spalte **Size** wird hexadezimal die Groesse jedes Segmentes angegeben. Weil eine Gruppe keine Groesse hat, wird nur das Wort GROUP eingetragen.

Die Spalte **Align** zeigt, mit welchem Typ einer Speicheradresse das Segment beginnt. Dieser Allign-Typ kann einer der folgenden sein:

- **byte**
- **word**
- **para**
- **page**
- **at**

Wenn das Segment nicht mit einem bestimmten Align-Typ definiert wurde, traegt MASM den Standard-Align-Typ fuer dieses Segment ein. Der Standardtyp ist PARA.

In der Spalte **Combine** wird der Typ des Segmentes ausgewiesen. Dieser Typ kann einer der folgenden sein:

- **none**
- **public**
- **stack**
- **memory**
- **common**
- **<adress>** (fuer den Align-Typ **at**)

Ist kein bestimmter Combine-Typ fuer das Segment definiert, wird NONE eingetragen. NONE stellt einen eigenen Combine-Typ dar. Ist der Align-Typ at, so enthaelt die Spalte Combine die hexadezimale Adresse fuer den Anfang des Segmentes.

Die Spalte **Class** beinhaltet den Klassennamen des Segmentes. Dieser Name wird genau so geschrieben wie in der Quelldatei, mit der Ausnahme, dass kleine Buchstaben in grosse gewandelt werden, wenn nicht der Schalter /ML angegeben war. Wurde kein Name spezifiziert, wird NONE eingetragen.

Die Align- und Combine-Typen und die Klassennamen werden im Abschnitt 3.4. ausfuehrlich erklaert.

10.4.5. Hinweise zum Lesen der Symboltabelle

Im folgenden Beispiel wird der Teil der Assemblerliste vorgestellt, der die Namen, Typen, Werte und Attribute aller Symbole der Quelldatei enthaelt.

Beispiel:

```
Symbols:

                N a m e         Type    Value   Attr

SYM0 . . . . . . . . . .        Number  0005
SYM1 . . . . . . . . . .        Text    1.234
SYM2 . . . . . . . . . .        Number  0008
SYM3 . . . . . . . . . .        Alias   SYM4
SYM4 . . . . . . . . . .        Text    5[BP][DI]
SYM5 . . . . . . . . . .        Opcode
SYM6 . . . . . . . . . .        L BYTE  002     DATA
SYM7 . . . . . . . . . .        L WORD  0012    DATA    Global
SYM8 . . . . . . . . . .        L DWORD 0022    DATA
SYM9 . . . . . . . . . .        L QWORD 0000            External
MARKE0 . . . . . . . . .        L FAR   0000            External
MARKE1 . . . . . . . . .        L NEAR  0010    CODE
```

Diese Tabelle hat vier Spalten:

N a m e Type Value Attr

In der Namensspalte sind alle Symbolnamen in alphabetischer Reihenfolge aufgelistet und so geschrieben, wie in der Quelldatei, mit der Ausnahme, dass kleine Buchstaben in grosse gewandelt werden, es sei denn, es wurde der /ML- bzw. /MX-Schalter angegeben. Namen, die laenger als 31 Zeichen sind, werden abgeschnitten.

Die Spalte **Type** zeigt den Typ des Symbols an. Er kann einer der folgenden sein:

Typ	Bedeutung
L NEAR	Marke vom Typ NEAR
L FAR	Marke vom Typ FAR
N PROC	Prozedurmarke vom Typ NEAR
F PROC	Prozedurmarke vom Typ FAR
Number	Absolutwert
Alias	ein "Alias-Name" fuer eine anderes Symbol
Opcode	Anweisungscode
Text	Speicheroperand, Zeichenfolge oder anderer Wert

Wurde ein Symbol durch die Pseudoperation EQU oder Gleichheitszeichen (=) definiert, erscheint in der Typ-Spalte entweder **Number**, **Opcode**, **Alias** oder **Text**. Ist das Symbol eine Variable, eine Marke oder eine Prozedur, wird in der Typ-Spalte die Symbollaenge angezeigt, falls sie bekannt ist. Die Laenge kann folgendermassen angegeben sein:

Typ	Laenge
BYTE	1 Byte (8 Bits)
WORD	1 Wort (16 Bits)
DWORD	1 Doppelwort (2 Worte)
QWORD	4 Worte
TBYTE	10 Bytes (5 Worte)
<number>	Laenge in Bytes einer strukturierten Variablen

Stellt das Symbol einen absoluten Wert dar, der mit der Pseudooperation EQU oder = definiert wurde, enthaelt die Spalte **Value** den Wert des Symbols. Dieser Wert kann ein anderes Symbol sein, eine Zeichenfolge oder ein konstanter numerischer Hexadezimalwert, abhaengig davon, ob der Typ Alias, Text oder Number ist. War der Typ Opcode, bleibt die Spalte leer. Fuer Variable, Marken oder Prozeduren enthaelt diese Spalte den Offset zum Beginn des Segmentes, in dem sie definiert ist.

In der Spalte **Attr** stehen die Attribute des Symbols. Diese Attribute beinhalten, wenn vorhanden, den Namen des Segments, den Gueltigkeitsbereich und die Kodelaenge. Ein Gueltigkeitsbereich ist nur dann gegeben, wenn das Symbol unter Verwendung von EXTRN oder PUBLIC definiert wurde. Dann ist der Gueltigkeitsbereich **External** oder **Global**. Die hexadezimale Kodelaenge wird nur fuer Prozeduren angegeben. Diese Spalte bleibt leer, wenn das Sumbol keine Attribute besitzt.

Wurde der DUP-Operator verwendet, erscheint in der Zeile eine zusaetzliche Laengenangabe **Length=<wert>**. <wert> ist eine hexadezimale Angabe und stellt den Wiederholungsfaktor vor dem DUP-Operator dar.

10.4.6. Hinweise zum Lesen einer Pass 1 - Liste

Wurde in der Befehlszeile von MASM der /D-Schalter spezifiziert, gibt der Assembler zusaetzlich eine Pass 1 - Liste in die Assemblerlistendatei aus. Diese zeigt das Ergebnis beider Paesse. Die Pass 1 - Liste kann helfen, die Ursache von Phasenfehlern zu finden.

Das folgende Beispiel erlaeutert die Pass 1 - Liste fuer eine ohne Fehler uebersetzte Quelldatei. Obwohl im Pass 1 ein Fehler auftritt, berichtigt MASM diesen Fehler im Pass 2 und beendet die Uebersetzung korrekt.

Waehrend des Pass 1 verursacht die jle-Anweisung mit einer Vorwaertsreferenz einen Fehler:

```
 0017  7E 00                        jle     smlstk
PASS_PROG.ASM (20) : error  9 : Symbol not defined SMLSTK
 0019  BB 1000                      mov     bx,4096
 001C                       smlstk:
```

MASM zeigt diesen Fehler an, weil er bis zu dieser Stelle noch nicht an die Definition des Symbols smlstk gekommen ist. Im Pass 2 ist smlstk definiert, so dass der Assembler die Anweisung ohne Fehler uebersetzen kann:

```
 0017  7E 03                        jle     smlstk
 0019  BB 1000                      mov     bx,4096
 001C                       smlstk:
```

Der Anweisungscode fuer jle enthaelt jetzt die 03 statt der 00. Das ist ein Sprung von 3 Byte.

Dort, wo MASM in beiden Paessen Code in der gleichen Bedeutung generiert, entsteht kein Phasenfehler. Ein Phasenfehler wird durch eine Meldung vom Assembler sichtbar gemacht.

In dem folgenden Programmfragment wird dies verdeutlicht. Eine falsch geschriebene Marke verursacht eine Phasenfehler.

```
 0000                       code    segment
 0000  E9 0000   U                  jmp     go
PASS_TEST.ASM(2) : error 9 : Symbol not defined go
 0003                       go      label   byte
 0003  B8 0001                      mov     ax,1
 0006                       code    ends
```

Die Marke go wurde in einer Vorwaertsreferenz verwendet und erzeugt im Pass 1 den Fehler **Symbol not defined.** Der Assembler nimmt an, dass das Symbol spaeter definiert wird, generiert 3 Bytes fuer den Kode und reserviert davon,2 Bytes fuer den aktuellen Wert des Symbols.

Im Pass 2 ist die Marke go bekannt als Marke vom Typ BYTE. Dieser Typ ist fuer die jmp-Anweisung nicht erlaubt. Als Ergebnis erzeugt MASM im Pass 2 nur 2 Bytes fuer den Kode, 1 Byte weniger als im Pass 1. Das ist die Ursache fuer einen Phasenfehler.

```
 0000                           code    segment
 0003  R                                jmp     go
PASS_TEST.ASM(2) : error 57 : Illegal size for item
 0003                           go      label   byte
PASS_TEST.ASM(3) : error 6 : Phase error between passes
 0003  B8 0001                          mov     ax,1
 0006                           code    ends
```

Die meisten Fehler aus dem Pass 1 werden im Pass 2 aufgeloest, so dass sie weder als Warnungen noch als schwere Fehler in Erscheinung treten. Konnten beispielsweise 5 Fehler aus dem Pass 1 waehrend des Pass 2 nicht aufgeloest werden, dann werden diese im Fehlerzaehler gezaehlt und auf der letzten Seite der Assemblerliste ausgewiesen, falls keine Pass-1 - Liste gefordert war.

Es gibt folgende 5 Phasenfehler:

Fehler-nummer	Meldung	Bedeutung
2	Register already defined	Register bereits definiert
4	Redefinition of symbol	Neudefinition eines Symbols
13	Must be declared in pass 1	Muss bereits im Pass 1 erklaert sein
17	Forward reference is illegal	Vorwaertsreferenz ist nicht erlaubt
85	End of file, no END directive	Dateiende, aber keine END-Anweisung

11. Crossreferenz

11.1. Einleitung

Das Programm **CREF** erzeugt eine Crossreferenzliste aller Symbole eines Assemblerprogramms.

Eine Crossreferenzliste ist eine alphabetische Aufzaehlung aller Symbole. Jedem Symbol folgt eine Anzahl von Zeilennummern. Diese Zeilennummer weist auf die Zeile im Quellprogramm, die einen Verweis auf das Symbol enthaelt.

CREF ist als Testhilfe zur Beschleunigung der Suche von Symbolen gedacht. Die Crossreferenz kann gemeinsam mit der vom Assembler erzeugten Symboltabelle den Test und die Korrektur von Programmen erleichtern.

11.2. Verwenden von CREF

CREF erzeugt eine Crossreferenzliste fuer ein Programm durch Umsetzen einer Nicht-ASCII-Crossreferenzdatei aus einem Assemblerlauf in eine lesbare ASCII-Datei.

Eine Crossreferenzdatei erhaelt man, indem man einen Namen fuer die Crossreferenzdatei beim Aufruf des Assemblers angibt. Eine Crossreferenzliste wird durch den Aufruf von CREF und die Angabe des Namens der Crossreferenzdatei aufgebaut.

In den Abschnitten 11.2. und 11.3. wird das ausfuehrlich erklaert.

11.2.1. Erzeugen einer Crossreferenzdatei

Eine Crossreferenzdatei kann durch die Eingabe eines Namens fuer diese Datei beim Aufruf von MASM erstellt werden. MASM bietet dazu 2 Wege:

- Dieser Dateiname kann die Antwort auf die entsprechende Eingabeaufforderung sein.
- Oder er kann in der Befehlszeile gemeinsam mit anderen Dateinamen eingegeben werden.

Bei der ersten Variante wird der Dateiname als Antwort auf die vierte Eingabeaufforderung eingegeben.

Es soll beispielsweise eine Crossreferenzdatei mit dem Namen test.crf fuer das Programm test.asm erstellt werden, dann sind folgende Eingaben notwendig

Beispiel:

```
MASM
Source filename [.ASM]: test
Object filename [test.OBJ]: test
Source listing [NUL.LST]: test
Cross-Reference [NUL.CRF]: test
```

Wird nach Cross-Reference [NUL.CRF]: kein Dateiname eingegeben, dann wird auch keine Crossreferenzdatei aufgebaut. Wurde keine Dateierweiterung angegeben, wird von MASM .CRF eingetragen. Diese Dateierweiterung erwartet CREF. Er wird fuer alle Crossreferenzdateien empfohlen.

Soll die Befehlszeile verwendet werden, so ist der Name der Crossreferenzdatei als vierter Parameter in der MASM-Befehlszeile zu plazieren.

Es soll beispielsweise eine Crossreferenzdatei mit dem Namen test.crf fuer die Quelldatei test.asm aufgebaut werden.

Beispiel:

MASM test,test,test,test

Mit dieser Befehlszeile werden ausserdem eine Objektkodedatei und eine Assemblerlistendatei waehrend der Uebersetzung erzeugt. Die einzelnen Parameter muessen durch Kommas getrennt werden. Sie sind Stellungsparameter, d. h. soll ein Parameter weggelassen werden, so ist stets das Komma einzugeben.

Weiter Informationen dazu enthaelt der Abschnitt 10.2.1.

11.2.2. Erzeugen einer Crossreferenzliste unter Verwendung von Eingabeaufforderungen

Das Programm CREF kann durch die Eingabe des Programmnamens gestartet werden. Danach erscheinen auf dem Bildschirm eine Reihe von Eingabeaufforderungen, die nach Dateinamen fragen. Folgende Schritte sind bei dieser Arbeitsweise notwendig:

1. Eingeben

   CREF

   und druecken <ENTER> in der DCP-Befehlsebene. Sobald CREF gestartet ist, wird angezeigt:

   Cross-Reference [.CRF]:

2. Eingeben des Namens der Crossreferenzdatei, die in eine Crossreferenzliste gewandelt werden soll und betaetigen von <ENTER>.

   Besitzt die Datei die Erweiterung .CRF, braucht keine Dateierweiterung angegeben werden. Hat sie eine andere Erweiterung, so muss diese exakt eingegeben werden.

   Danach ist der Dateinamen fuer die Crossreferenzliste einzugeben.

   Listing [dateiname.REF]:

   Beachte, dass <dateiname> der Standarddateiname fuer die Crossreferenzliste ist.

3. <ENTER> eingeben, wenn der Standarddateiname verwendet werden soll. Sonst ist der Dateiname und <ENTER> einzugeben. Wenn keine Dateierweiterung eingegeben wurde, wird .REF eingesetzt.

Sobald dieser Dateiname eingegeben wurde, liest CREF die Crossreferenzdatei und erzeugt eine neue Liste. Die Anzahl der Symbole in der Crossreferenzdatei wird angezeigt.

Beispiel:

```
CREF
Cross Reference                Version x.xx


Cross reference [.CRF]: test
Listing [test.REF]:

8 Symbols
```

In diesem Beispiel wird test.crf gelesen und das Ergebnis in test.ref ausgegeben. Die Crossreferenz enthaelt 8 Symbole.

11.2.3. Erzeugen einer Crossreferenzliste unter Verwendung der Befehlszeile

Man kann eine Crossreferenzliste auch durch die Eingabe von **CREF**, gefolgt von den Namen der zu verarbeitenden Dateien, erzeugen.

Die Befehlszeile hat folgende Form:

CREF <crossreferenzdatei>[,<crossreferenzliste>][;]

<crossreferenzdatei> bezeichnet den Namen der durch MASM erzeugten Crossreferenzdatei und <crossreferenzliste> ist der Name der zu erstellenden lesbaren ASCII-Datei.

Wurden keine Dateierweiterungen angegeben, nimmt CREF automatisch .CRF fuer die Crossreferenzdatei und .REF fuer die Crossreferenzliste an.

Durch die Eingabe eines Semikolons (;) nach dem Namen der Crossreferenzdatei wird der Standarddateiname fuer die Liste ausgewaehlt. Dieser Standarddateiname hat den selben Dateinamen wie die Crossreferenzdatei und die Dateierweiterung .REF.

Vor einer der beiden Dateien kann ein Laufwerk oder ein Pfad angegeben werden. Ebenso ist ein Geraetename (z. B. **CON** oder **PRN**) fuer die Ausgabe moeglich.

Beispiele:

1. **CREF test.crf,test.ref**

In diesem Beispiel wird die Crossreferenzdatei test.crf verwendet, um die Crossreferenzliste test.ref zu erzeugen. Das Gleiche wird auch erreicht durch:

2. **CREF test,test**

oder

3. **CREF test;**

In dem folgenden Beispiel wird die Crossreferenzliste auf dem Bildschirm angezeigt. Es wird keine Datei ausgegeben.

4. **CREF test,con**

11.3. Das Format einer Crossreferenzliste

Die Crossreferenzliste enthaelt die Namen aller im Progamm vorkommenden Symbole. Jedem Namen folgt eine Liste von Zeilennummern, die Zeilen in der Assemblerliste angeben, in denen das Symbol entweder definiert oder verwendet wurde. Die auf die Definition des Symbols verweisende Zeilennummer ist durch ein Nummernzeichen (#) markiert.

Jede Seite dieser Liste beginnt mit dem Titel des Programms. Der Titel ist der Name oder die Zeichenkette, die durch die Pseudooperation TITLE in der Quelldatei definiert wurde. (Siehe dazu auch den Abschnitt 9.6.)

Beispiel:

Wir nehmen an, dass das folgende Beispiel in der Datei test2.asm ist:

```
quit    MACRO
        ;Rueckkehr zu DCP
        mov     ah,4Ch
        ;;DCP-Endefunktion
        int     21h
        ENDM

max     EQU     65535

        EXTRN   work:NEAR

stack   SEGMENT para public 'STACK'
        DB      256 DUP(?)
stack   ENDS

data    SEGMENT public 'DATA'
puffer  DW      100 DUP(?)
data    ENDS

code    SEGMENT public 'CODE'
ASSUME  cs:code,ds:data

start:  mov     ax,data
        ;Ladeadresse
        mov     ds,ax
        call    work
        ;Unterprogrammaufruf
        quit                ;Makroaufruf
code    ENDS
        END     start
```

Das Programm wird uebersetzt und es wird eine Crossreferenzdatei erzeugt durch die Eingabe von:

```
MASM test2,test2,test2,test2
```

Die bei diesem Assemblerlauf entstandene Liste wird im folgenden gezeigt:

```
MASM (DCPX) V1.0                                    5/5/87 06:42:57

                                                       Page     1-1

                                        quit   MACRO
                                               ;Rueckkehr zu DCP
                                               mov      ah,4Ch
                                               ;;DCP-Endefunktion
                                               int      21h
                                               ENDM

= FFFF                                  max    EQU      65535

                                               EXTRN    work:NEAR

0000                              stack SEGMENT para public 'STACK'
0000   0100[                                   DB       256 DUP(?)
            ??
                                  ]

0100                                    stack ENDS

0000                                    data   SEGMENT public 'DATA'
0000   0064[                            puffer DW       100 DUP(?)
            ????
                                  ]

00C8                                    data   ENDS

0000                                    code   SEGMENT public 'CODE'
                                               ASSUME cs:code,ds:data

0000   B8 ---- R                        start:mov       ax,data
                                               ;Ladeadresse
0003   8E D8                                   mov      ds,ax
0005   E8 0000 E                               call     work
                                               ;Unterprogrammaufruf
                                               quit
0008   B4 4C                          1        mov      ah,4Ch
000A   CD 21                          1        int      21h
                                               ;Makroaufruf
000C                                    code   ENDS
                                               end      start
```

```
MASM (DCPX) V1.0                                5/5/87 06:42:57

                                                        Symbols-1

Macros:

                N a m e                 Lines

QUIT . . . . . . . . . . . . . . . .    3

Segments and Groups:

                N a m e                 Size  Align   Combine Class

CODE . . . . . . . . . . . . . . . .    000C  PARA    PUBLIC 'CODE'
DATA . . . . . . . . . . . . . . . .    00C8  PARA    PUBLIC 'DATA'
STACK  . . . . . . . . . . . . . . .    0100  PARA    PUBLIC 'STACK'

Symbols:

                N a m e                 Type  Value   Attr

MAX  . . . . . . . . . . . . . . . .    Number FFFF

PUFFER . . . . . . . . . . . . . . .    L WORD 0000   DATA
                                                      Length = 0064

START  . . . . . . . . . . . . . . .    L NEAR 0000   CODE

WORK . . . . . . . . . . . . . . . .    L NEAR 0000   External


     31 Source  Lines
     34 Total   Lines
     29 Symbols

  49812 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors
```

Um die Crossreferenzliste als Datei test2.crf zu erzeugen, ist

CREF test2,test2

einzugeben. Diese entstandene Datei test2.ref ist folgendermassen aufgebaut:

```
CREF (DCPX) V1.0


  Symbol Cross-Reference    (# is definition)              Cref-1

CODE . . . . . . . . . . . . 26    26#   26    27    38

DATA . . . . . . . . . . . . 19    19#   19    24    27    29

MAX. . . . . . . . . . . . . 8     8#

PUFFER . . . . . . . . . . . 20    20#

QUIT . . . . . . . . . . . . 34

STACK. . . . . . . . . . . . 12    12#   12    17
START. . . . . . . . . . . . 29    29#   39

WORK . . . . . . . . . . . . 10    10#   32
```

Die Zeilennummern in der Crossreferenzliste und in der Assemblerliste lassen sich vergleichen. Man sollte nicht versuchen, die Zeilen in der Quelldatei zu zaehlen, da diese Zeilennummern nicht zu denen in der Assemblerliste und in der Crossreferenzdatei passen.

Anhang A

MASM-Fehlermeldungen

In diesem Anhang werden die Fehlermeldungen und Warnungen von MASM aufgelistet und kurz erklaert.

Fehlermeldungen werden sofort ausgegeben, wenn ein Fehler waehrend der Uebersetzung auftritt. Warnungen sind das Ergebnis fragwuerdiger Anweisungen.

Nach der Abarbeitung wird eine Endemeldung angezeigt, auch wenn kein Fehler aufgetreten ist. Diese Meldung enthaelt den freien Platz in der Symboltabelle in Bytes und die Anzahl Fehler und Warnungen.

Wurde der /V-Schalter gesetzt, werden zusaetzlich die Anzahl Quellzeilen, die Gesamtzahl Zeilen (einschliesslich der Makroerweiterungen), sowie die Anzahl der Symbole angezeigt.

Beispiel:

```
      1108 Source  Lines
      1286 Total   Lines
       215 Symbols

     44814 Bytes symbol space free

         0 Warning Errors
         0 Severe  Errors
```

Die ersten 3 Zeilen werden nur durch den /V-Schalter auf dem Bildschirm angezeigt.

Die Meldungen sind in der Reihenfolge ihrer Nummern aufgefuehrt. Wenn erforderlich, sind Verweise auf Abschnitte der MASM-Beschreibung gegeben.

0 Block nesting error

Geschachtelte Prozeduren, Segmente, Strukturen, Makros, IRC-, IRP- oder REPT-Anweisungen sind nicht richtig beendet worden. Dieser Fehler tritt beispielsweise auf, wenn die aeussere Anweisung abgeschlossen wird, obwohl die innere noch eroeffnet ist.

1 Extra characters on line

Ausser den fuer die Anweisung notwendigen Zeichen sind darueber hinaus ueberfluessige Zeichen auf einer Zeile enthalten.

3 Unknown symbol type

MASM erkennt die in einer LABEL- oder EXTRN-Anweisung spezifizierte Typgroesse nicht an.

Beispiel:

```
marke   LABEL   bite
```

Es ist ein gueltiger Typ, wie BYTE, WORD, NEAR usw. einzutragen.

4 Redefinition of symbol

Wurde ein Symbol zweimal definiert, erscheint dieser Fehler im 1. Pass bei der zweiten Definition (siehe auch die Fehler 5 und 26).

5 Symbol is multi-defined

Wurde ein Symbol zweimal definiert, erscheint dieser Fehler im Pass2 bei beiden Definitionen (siehe auch Fehler 4 und 26).

6 Phase error between passes

Das Programm enthaelt eine zweideutige Anweisung, so dass eine Marke ihren Wert zwischen Pass 1 und Pass 2 aendert. Das kann beispielsweise bei Vorwaertsreferenzen ohne den erforderlichen Praefix auftreten. Mittels /D-Schalter kann eine Pass 1 - Liste erzeugt werden, die beim Aufloesen der Phasenfehler hilft.

7 Already had ELSE clause

Es wurde eine ELSE-Anweisung innerhalb einer existierenden ELSE-Klausel definiert. Zu jedem IF ist nur eine ELSE-Anweisung moeglich.

8 Not in conditional block

Ein ENDIF oder ELSE wurde spezifiziert, ohne vorher eine bedingte Pseudooperation zu aktivieren.

9 Symbol not defined

Es wurde ein undefiniertes Symbol verwendet. Eine moegliche Ursache fuer diesen Fehler wird im Abschnitt 10.4.6. erklaert.

10 Syntax error

Syntaxfehler

11 Type illegal in context

In diesem Zusammenhang nicht erlaubter Typ

12 Should have been group name

Es muss ein Gruppenname spezifiziert werden

13 Must be declared in pass 1

Das Symbol muss vor seiner Verwendung definiert sein.

14 Symbol type usage illegal

Unerlaubte Verwendung eines PUBLIC-Symbols (siehe Abschnitt 6.2.)

15 Symbol already different kind

Versuch, ein Symbol zu definieren, das schon in anderer Art vorhanden ist.

16 Symbol is reserved word

Versuch, ein fuer den Assembler reserviertes Wort unerlaubt zu benutzen (zum Beispiel mov als Variable).

17 Forward reference is illegal

Unerlaubte Vorwaertsreferenz

Beispiel:

```
DB      zaehler DUP(?)
zaehler EQU     10
```

In umgekehrter Reihenfolge werden die Anweisungen fehlerfrei uebersetzt.

18 Must be register

Der Operand muss ein Register sein, aber es wurde ein Symbol angegeben.

19 Wrong type of register

Der Befehl erwartet einen bestimmten Registertyp. Es wurde ein ungueltiges Register angegeben.

Beispiel:

```
INC    CS
```

20 Must be segment or group

Es muss ein Segment oder eine Gruppe sein.

21 Symbol has no segment

Eine Variable wurde mit SEG verwendet, obwohl sie kein bekanntes Segment hat.

22 Must be symbol type

Es muss der Typ WORD, DW, ØW, BYTE oder aehnliches spezifiziert werden.

23 Already defined locally

Es wurde versucht, ein Symbol als EXTRN zu erklaeren, welches schon lokal definiert ist.

24 Segment parameters are changed

Die Argumente der SEGMENT-Anweisung sind nicht identisch mit denen einer vorherigen SEGMENT-Definition fuer das gleiche Segment.

25 Not proper align/combine type

Die Parameter der SEGMENT-Anweisung sind nicht korrekt. Der ALIGN- und COMBINE-Typ ist zu pruefen (siehe Abschnitt 3.4.)

26 Reference to mult defined

Die Anweisung enthaelt ein mehrfach definiertes Symbol (siehe Fehler 4 und 5).

27 Operand was expected

Der Assembler erwartet einen Operanden, aber es wurde ein Operator angegeben.

28 Operator was expected

Der Assembler erwartet eine Operator, aber es wurde ein Operand angegeben.

29 Division by Ø or overflow

Der Ausdruck enthaelt eine Division durch Ø, oder eine Zahl ist laenger als darstellbar.

3Ø Shift count is negative

Der Verschiebeausdruck enthaelt einen negativen Verschiebefaktor.

31 Operand types must match

Der Assembler hat an einer Stelle verschiedene Arten oder Groessen von Argumenten erhalten, an der sie angepasst sein muessen.

Beispiel:

```
mov     ax,bh
```

Der Befehl ist nicht erlaubt. Beide Operanden muessen entweder Wort oder Byte sein.

32 Illegal use of external

Nicht erlaubte Verwendung eines Externals

Beispiel:

```
DB      zaehler DUP(?)
```

ist nicht erlaubt, wenn zaehler extern definiert ist (siehe Abschnitt 6.3.)

33 Must be record field name

Es muss ein Recordfeldname spezifiziert werden.

34 Must be record or field name

Es muss ein Recordname oder ein Recordfeldname eingetragen werden.

35 Operand must have size

Der Operand muss eine Groesse besitzen. Dieser Fehler kann meistens durch Verwenden des PTR-Operators vermieden werden.

36 Must be var, label or constant

Eine Variable, Marke oder Konstante muss angegeben werden.

37 Must be structure field name

Es wird ein Strukturfeldname erwartet.

38 Left Operand must have segment

Im rechten Operanden wurde etwas verwendet, das im linken Operanden eine Segmentangabe erfordert.

Beispiel:

```
:symbol     ; ist ungueltig
seg:symbol  ; waere richtig
```

39 One Operand must be const

Der Additionsoperator wurde unerlaubt verwendet (siehe Abschnitt 5.3.1.).

40 Operands must same or 1 abs

Der Subtraktionsoperator wurde unerlaubt verwendet (siehe Abschnitt 5.3.1.).

41 Normal type operand expected

Es wurde STRUC, BYTE, WORD oder ein anderer ungueltiger Operand angegeben. Erwartet wird eine variable Marke.

42 Constant was expected

Der angegebene Ausdruck hat als Ergebnis keine Konstante, zum Beispiel ein Variablenname oder ein External.

43 Operand must have segment

Unerlaubte Verwendung des SEG-Operators (siehe Abschnitt 5.3.12.).

44 Must be associated with data

Es wurde ein koderelativer Ausdruck geschrieben, aber ein datenrelativer Ausdruck wird erwartet.

Beispiel:

```
marke:  mov     ax,LENGTH ds:marke
```

Diese Zeile versucht einen Ausdruck mittels DS zu adressieren, waehrend der Ausdruck in Wirklichkeit durch CS adressiert wird.

45 Must be associated with code

Es wurde ein datenrelativer Ausdruck geschrieben, aber ein koderelativer wird erwartet.

Beispiel:

```
jmp test
```

Dieser Fehler erscheint fuer diese Zeile, wenn test im Datensegment definiert ist.

46 Already have base register

In einem Operanden wurde mehr als ein Basisregister angegeben.

Beispiel:

```
mov     ax,[bx+bp]
```

47 Already have index register

In einem Operanden wurde mehr als ein Indexregister verwendet.

Beispiel:

```
mov     ax,[si+di]
```

48 Must be index or base register

Der Befehl fordert ein Index- oder Basisregister. Es wurde ein anderes Register in den eckigen Klammern ([]) geschrieben.

Beispiel:

```
mov     ax,[bx+ax]
```

49 Illegal use of register

Unerlaubte Verwendung eines Registers

50 Value ist out of range

Der angegebene Wert ist zu gross.

51 Operand not in IP segment

Auf einen Operanden kann nicht zugegriffen werden, weil er sich nicht im aktuellen IP-Segment befindet.

52 Improper operand type

Es wurde ein Operand so benutzt, dass kein Operationskode generiert werden konnte.

53 Relative jmp out of range

Bedingte Spruenge muessen im Bereich von -128 bis +127 vom aktuellen Befehl ausgefuehrt werden. Der spezifizierte Sprung liegt ausserhalb dieses Bereichs. Die Korrektur erfolgt durch Umkehren der Bedingung im bedingten Sprung und dem Ansprung der ausserhalb liegenden Marke durch einen unbedingten Sprung (JMP).

54 Index displ. must be constant

Unerlaubte Verwendung einer Indexverschiebung

55 Illegal register value

Der spezifizierte Registerwert passt nicht in das "reg"-Feld des Befehls (der Wert ist groesser als 7).

56 No immediate mode

Es wurde ein Direktwert fuer einen Befehl spezifiziert, der keinen Direktwert akzeptieren kann.

Beispiel:

```
mov     ds,data
```

Diese Anweisung ist nicht erlaubt. Die Segmentadresse muss erst in ein allgemeines Register und dann nach DS transportiert werden.

57 Illegal size for item

Die Groesse der Bezugnahme im Ausdruck ist nicht erlaubt. Ein Beispiel fuer diesen Fehler wird im Abschnitt 10.4.6. gezeigt. Der Fehler kann durch Verwenden des PTR-Operators beseitigt werden (siehe Abschnitt 5.3.6. und 5.5.).

58 Byte register is illegal

In einem Ausdruck wurde eines der Byteregister verwendet, wo es nicht erlaubt ist.

Beispiel:

```
PUSH    AL    ; ist nicht erlaubt
PUSH    AX    ; ist richtig
```

59 CS register illegal usage

Unerlaubte Verwendung des Registers CS.

Beispiel:

```
XCHG    CS,AX
```

60 Must be AX or AL

Der Befehl erlaubt nur die Register AX oder AL. Es wurde ein anderes Register spezifiziert.

Beispiel:

IN-Befehl

61 Improper use of segment reg

Ein Segmentregister wurde in einem Befehl unerlaubt verwendet.

Beispiel:

Transport eines Direktwertes in ein Segmentregister

62 No or unreachable CS

Es wurde versucht an eine unerreichbare Marke zu springen.

63 Operand combination illegal

Die verwendete Operandenkombination ist in diesem 2-Operanden-Befehl nicht erlaubt.

64 Near JMP/CALL to different CS

Es wurde ein Sprung oder ein Unterprogrammaufruf innerhalb eines Segments (NEAR) an eine Stelle in einem anderen Kodesegment, das durch ein anderes ASSUME CS definiert ist, versucht.

65 Label can't have seg. override

Der Segmentpraefix wurde unerlaubt benutzt (siehe Abschnitt 5.3.7.).

66 Must have opcode after prefix

Einer der Praefixe REPE, REPNE, REPZ oder REPNZ wurde ohne nachfolgenden Operationskode verwendet.

67 Can't override ES segment

Das Extrasegment wurde in einem Befehl unerlaubt ueberschrieben.

Beispiel:

```
STOS    DS:TARGET  ; ist nicht erlaubt
```

68 Can't reach with segment reg

Es fehlt die ASSUME-Anweisung, die diese Variable adressierbar macht.

69 Must be in segment block

Es wurde versucht, Kode ausserhalb eines Segments zu generieren.

70 Can't use EVEN on Byte segment

Die EVEN-Anweisung wurde in einem Segment mit Byteausrichtung verwendet (siehe Abschnitt 3.9.).

72 Illegal value for DUP count

Der Zaehler des DUP-Operators muss eine konstante positive ganze Zahl groesser Null sein.

73 Symbol already external

Es wird versucht, ein externes Symbol lokal zu definieren.

74 DUP is too large for linker

Die Schachtelungstiefe der DUP-Operatoren ist zu gross (siehe Abschnitt 4.3.6.).

75 Usage of ? (indeterminate) bad

Falsche Verwendung des undefinierten Operanden (?).

Beispiel:

```
? + 5  ; ist nicht erlaubt
```

76 More values than defined with

Fuer eine durch RECORD oder STRUC definierte Variable wurden zu viele Initialisierungswerte angegeben.

77 Only initialize list legal

Es wurde ein Strukturname ohne spitze Klammern (<>) verwendet.

78 Direktive illegal in STRUC

Die Definition eines Strukturtyps darf nur Felddefinitionen und Kommentare enthalten. Andere Anweisungen sind nicht erlaubt (siehe Abschnitt 4.5.1.).

79 Override with DUP is illegal

Der DUP-Operator ist als Initialisierungswert in einer Strukturvereinbarung nicht erlaubt.

80 Field cannot be overriden

Das Feld der Strukturvereinbarung kann nicht ueberschrieben werden.

81 Override is of wrong type

Der Initialisierungswert fuer das Feld einer Strukturvereinbarung hat die falsche Groesse.

Beispiel:

Mit dem mit DW definierten "MELDUNG" soll ein mit DB definierter Standardwert ueberschrieben werden.

83 Circular chain of EQU aliases

Eine EQU-Anweisung weist eventuell auf sich selbst.

84 8087 opcode can't be emulated

Entweder der vom Befehl verwendete 8087-Operationskode oder die Operanden koennen vom Emulator nicht unterstuetzt werden.

85 End of file, no END direktive

Die END-Anweisung wurde vergessen oder es liegt ein Fehler bei Verschachtelungen vor.

86 Data emitted with no segment

Daten oder Kode generierende Anweisungen liegen ausserhalb eines Segmentes.

Beispiel:

```
code    SEGMENT
          .
          .
          .
code    ENDS
        push    ax
test    DW      ?
        END
```

Beide vor dem END stehenden Anweisungen wuerden diesen Fehler verursachen. Jede Daten oder Kode generierende Anweisung muss innerhalb eines Segments stehen.

87 Forced error - pass 1

Es wurde mittels .ERR1 ein Fehler erzwungen

88 Forced error - pass 2

Es wurde mittels .ERR2 ein Fehler erzwungen

89 Forced error

Es wurde mittels .ERR ein Fehler erzwungen

90 Forced error - expression equals 0

Es wurde mittels .ERRE ein Fehler erzwungen

91 Forced error - expression not equal 0

Es wurde mittels .ERRNZ ein Fehler erzwungen

92 Forced error - symbol not defined

Es wurde mittels .ERRNDEF ein Fehler erzwungen

93 Forced error - symbol defined

Es wurde mittels .ERRDEF ein Fehler erzwungen

94 Forced error - string blank

Es wurde mittels .ERRB ein Fehler erzwungen

95 Forced error - string not blank

Es wurde mittels .ERRNB ein Fehler erzwungen

96 Forced error - string identical

Es wurde mittels .ERRIDN ein Fehler erzwungen

97 Forced error - string different

Es wurde mittels .ERRDIF ein Fehler erzwungen

98 Override value is wrong length

Der ueberschreibende Wert fuer ein Strukturfeld ist zu lang. Er passt nicht in das Feld.

Beispiel:

```
x       STRUC
x1      DB      "A"
x       ENDS
y       x       <"AB">
```

Der ueberschreibende Wert ist eine zwei Bytes lange Kette. Die Strukturtypdefinition sieht nur Platz fuer ein Byte vor.

99 Line to long expanding symbol

Ein mit EQU oder = definiertes Symbol ist so lang, dass seine Erweiterung ein Ueberlaufen des internen Assemblerpuffers verursacht. Diese Meldung kann ein rekursives Textmakro anzeigen.

100 Impure memory reference

Der Kode enthaelt den Versuch, Daten ins Kodesegment zu schreiben, wenn .8286p und der /P-Schalter wirken.

Beispiel:

```
code    SEGMENT
        ASSUME  CS:code
c_word  DW      ?
          .
          .
          .
        mov     cs:c_word,data
code    ENDS
```

Der /P-Schalter prueft Anweisungen, die im nichtgeschuetzten Modus erlaubt sind und im geschuetzten Modus Probleme verursachen koennen.

101 Missing data; zero assumed

Fuer eine Anweisung fehlt ein Operand.

Beispiel:

```
mov     ax,
```

Der Kode wird uebersetzt, als stuende

```
mov     ax,0.
```

Das ist nur eine Warnung. Die Objektkodedatei wird nicht geloescht.

102 Segment near (or at) 64k limit

Das ist eine Warnung. Die Objektkodedatei wird nicht geloescht. Diese Warnung tritt nur fuer Kodesegmente nicht fuer Datensegmente auf, wenn sie bis auf 36 Bytes die maximale Groesse von 64k erreicht haben. Sie wird ausgegeben, wenn das Segment mit ENDS abgeschlossen wird. Ein Kodesegment ist ein Segment, das mindestens einen Befehl enthaelt.

Zusaetzlich gibt der Assembler eine Meldung ohne Nummer aus:

Out of Memory

Aller verfuegbare Speicher wurde aufgebraucht, entweder weil die Quelldatei zu gross ist, oder weil zu viele Symbole definiert wurden.

Es gibt verschiedene Moeglichkeiten, diesen Fehler zu beseitigen. Zunaechst sollte man versuchen, ohne Objektkodedatei zu uebersetzen. Wenn das geht, assembliert man, um eine Assembleliste oder eine Crossreferenz zu erhalten. Dabei spezifiziert man keine Objektkodedatei. Danach muss die Quelldatei geaendert werden, um den benoetigten Platz fuer die Symbole zu verringern. Dabei ist folgendes zu beachten:

- minimale Verwendung von Makros, Strukturen, EQU- und = - Anweisungen
- kurze Symbolnamen verwenden
- Tabulatoren statt mehrere Leerzeichen in Makros verwenden
- Makrokommentar mit zwei statt einem Semikolon kennzeichnen
- nicht mehr benoetigte Makrodefinitionen loeschen.

Anhang B

Uebersicht ueber die Pseudooperationen des Assemblers

.186 123

Erlaubt den Befehlssatz der Prozessoren K1810 WM86, 8086 und 80186

.286c 123

Erlaubt den Befehlssatz des nichtgeschuetzten Modus fuer den 80286

.286p 123

Erlaubt den Befehlssatz der Prozessoren K1810 WM86, 8086 und alle Befehle des 80286

.287 123

Erlaubt den Befehlssatz der Arithmetikprozessoren 8087 und 80287

.8086 123

Erlaubt den Befehlssatz der Prozessoren K1810 WM86, 8086 und 8088

.8087 123

Erlaubt nur den Befehlssatz des Arithmetikprozessors 8087

<name> = <ausdruck> 147

Absolutes Symbol durch Zuweisen des numerischen Wertes von <ausdruck> an <name> erzeugen

ASSUME <segmentregister>:<segmentname>
[,<segmentregister>:<segmentname>,...] 134

Weist <segmentregister> als Standard allen Marken und Variablen in dem durch <segmentname> gegebenen Segment oder Gruppe zu

COMMENT <begrenzer>
<text>
<begrenzer> [<text>] 120

Betrachtet <text> zwischen den begrenzenden Zeichen <begrenzer> als Kommentar

.CREF 220

Hebt die Wirkung von .XCREF auf

[<name>] DB <initwert>[,<initwert>,...] 139

Zuweisen eines Bytes des Speichers und initialisieren

[<name>] DW <initwert>[,<initwert>,...] 140

Zuweisen eines Wortes (2 Bytes) des Speichers und initialisieren

[<name>] DD <initwert>[,<initwert>,...] 142

Zuweisen eines Doppelwortes (4 Bytes) des Speichers und initialisieren

[<name>] DQ <initwert>[,<initwert>,...] 143

Zuweisen von 8 Bytes des Speichers und initialisieren

[<name>] DT <initwert>[,<initwert>,...] 144

Zuweisen von 10 Bytes des Speichers und initialisieren

ELSE 188

Die dem ELSE folgenden Anweisungen werden bei nicht erfuellter Bedingung der zugehoerigen bedingten Pseudooperation IFxxxx uebersetzt (xxxx steht fuer den entsprechenden Bedingungskode).

END 131

Kennzeichnet das Ende einer Quelldatei, <ausdruck> stellt den Eintrittspunkt in das Hauptprogramm dar.

ENDIF 188

Beendet einen bedingten Block

ENDM 198,203,204,205

Beendet eine Makrodefinition oder einen Wiederholungsblock

<name> ENDP 136

Bezeichnet das Ende einer Prozedur

<name> ENDS 124,149

Kennzeichnet das Ende eines Segments bzw. das Ende einer Strukturdefinition

<name> EQU <ausdruck> 147

Erzeugen von absoluten Symbolen, Aliasnamen oder Textsymbolen durch Zuweisen von <ausdruck> an <name>

.ERR 193

Generiert einen Fehler

.ERR1 193

Generiert nur im Pass 1 einen Fehler

.ERR2 193

Generiert nur im Pass 2 einen Fehler

.ERRB <zeichenkette> 195

Generiert einen Fehler, wenn <zeichenkette> leer ist

.ERRDEF <name> 194

Generiert einen Fehler, wenn <name> als Marke, Variable oder Symbol definiert ist

.ERRDIF <zeichenkette1>,<zeichenkette2> 196

Generiert einen Fehler, wenn <zeichenkette1> und <zeichenkette2> verschieden sind.

.ERRE <ausdruck> 194

Generiert einen Fehler, wenn <ausdruck> falsch (Null) ist

.ERRIDN <zeichenkette1>,<zeichenkette2> 196

Generiert einen Fehler, wenn <zeichenkette1> und <zeichenkette2> identisch sind

.ERRNB <zeichenkette> 195

Generiert einen Fehler, wenn <zeichenkette> nicht leer ist.

.ERRNDEF <name> 194

Generiert einen Fehler, wenn <name> bis zu dieser Stelle noch nicht definiert worden ist.

.ERRNZ <ausdruck> 194

Generiert einen Fehler, wenn <ausdruck> wahr (nicht Null) ist

EVEN 135

Richtet das naechste Daten- oder Anweisungsbyte an einer Wortgrenze aus

EXITM 206

Beendet die Erweiterung eines Makros oder Wiederholungsblockes vorzeitig

EXTRN <name>:<typ>[,<name>:<typ>,...] 184

Definiert <name> als externe Bezugnahme mit dem angegebenen <typ>

<name> GROUP <segmentname>[,<segmentname>,...] 132

Vergibt einem oder mehreren Segmenten einen Gruppennamen

IF <ausdruck> 188

Erlaubt die Uebersetzung der nachfolgenden Anweisungen, wenn <ausdruck> wahr ist

IF1 189

Erlaubt die Uebersetzung der nachfolgenden Anweisungen nur im Pass 1

IF2 189

Erlaubt die Uebersetzung der nachfolgenden Anweisungen nur im Pass 2

IFB <argument> 190

Erlaubt die Uebersetzung der nachfolgenden Anweisungen bei leerem <argument>

IFDEF <name> 189

Erlaubt die Uebersetzung der nachfolgenden Anweisungen, wenn <name> eine Marke, Variable oder ein Symbol ist

IFDIF <argument1>,<argument2> 191

Erlaubt die Uebersetzung der nachfolgenden Anweisungen, wenn <argument1> und <argument2> verschieden sind

IFE <ausdruck> 188

Erlaubt die Uebersetzung der nachfolgenden Anweisungen, wenn der Wert von <ausdruck> falsch (Null) ist

IFIDN <argument1>,<argument2> 191

Erlaubt die Uebersetzung der nachfolgenden Anweisungen, wenn <argument1> und <argument2> identisch sind

IFNB <argument> 190

Erlaubt die Uebersetzung der nachfolgenden Anweisungen bei nicht leerem <argument>

IFNDEF <name> 189

Erlaubt die Uebersetzung der nachfolgenden Anweisungen, wenn <name> nicht definiert ist

INCLUDE <dateiname> 212

Fuegt Quellkode aus der durch <dateiname> spezifizierten Datei in die aktuelle Datei ein

IRP <symbolischer name>,<[<paramter>[,<parameter>,...]]> 204

Beginn eines Wiederholungsblockes; die Wiederholung erfolgt fuer jeden <parameter>

IRPC <symbolischer name>,<zeichenkette> 205

Beginn eines Wiederholungsblockes; die Wiederholung erfolgt fuer jedes Zeichen in <zeichenkette>

<name> LABEL <typ> 148

Erzeugen einer Variablen oder Marke durch Zuweisen des Wertes des aktuellen Speicherplatzzaehlers und des durch <typ> gegebenen Typs an <name>

.LALL 218

Alle Quellanweisungen eines Makros, einschliesslich Kommentare mit einem einfachen Semikolon (;) werden aufgelistet

.LFCOND 217

Laesst das Listen der falschen bedingten Bloecke wieder zu

.LIST 217

Hebt die Wirkung von .XLIST auf; nachfolgende Quellanweisungen werden wieder in die Liste aufgenommen

LOCAL <dummyname>[,<dummyname>,...] 201

erzeugt Symbolnamen innerhalb eines Makros

<name> MACRO [<dummyparameter>[,<dummyparameter>,...]] 198

Beginn einer Makrodefinition

NAME <modulname> 214

Gibt dem aktuellen Modul den Namen <modulname>

ORG <ausdruck> 134

Setzt den Speicherplatzzaehler im aktuellen Segment auf <ausdruck>

%OUT [<text>] 214

Zeigt den spezifizierten <text> waehrend der Uebersetzung auf dem Bildschirm an

PAGE <laenge>,<breite> 216

Stellt die Seitenlaenge und die Zeilenbreite fuer die Assemblerliste ein

PAGE + 216

Erhoeht in der Assemblerliste die Abschnittsnummer um 1 und setzt die Seitennummer auf 1 zurueck

PAGE 216

Vorschub auf die neue Seite in der Assemblerliste

<name> PROC [<reichweite>] 136,138

Kennzeichnet den Beginn einer Prozedur und die <reichweite> NEAR oder FAR

PUBLIC <name>[,<name>,...] 184

Definiert <name> als Eintrittspunkt (globales Symbol)

PURGE <makroname>[,<makroname>,...] 203

Loescht Makrodefinitionen

.RADIX <ausdruck> 212

Setzt die Eingabezahlenbasis auf <ausdruck>

<recordname> RECORD <feldname>:<anzahl>[=<ausdruck>]
[,<feldname>:<anzahl>[=<ausdruck>],...] 151

Definiert einen Recordtyp

REPT <ausdruck> 203

Beginn eines Wiederholungsblockes; <ausdruck> gibt die Zahl der Wiederholungen an

.SALL 218

Unterdrueckt das Listen aller Makroerweiterungen

<name> SEGMENT [<align>] [<combine>] ['<class>'] 124

Markiert den Beginn eines Programmsegments genannt <name> mit den Attributen <align> und <class>.

.SFCOND 217

Unterdrueckt das Auflisten aller nachfolgenden falschen bedingten Bloecke

<name> STRUC 149

Kennzeichnet den Beginn einer Strukturtypdefinition

SUBTTL [<text>] 215

spezifiziert einen Untertitel fuer die Assemblerliste

.TFCOND 217

Schaltet das Listen der falschen bedingten Bloecke um

TITLE <text> 215

Spezifiziert eine Titelzeile fuer die Assemblerliste

.XALL 218

Nur Code oder Daten generierende Anweisungen in Makros werden aufgelistet

.XCREF [<name>[,<name>,...]] 220

Unterdrueckt das Erzeugen einer Crossreferenz fuer die angegebenen Namen

.XLIST 217

Nachfolgende Quellzeilen werden nicht in die Assembler-liste aufgenommen.

Anhang_C

Uebersicht_ueber_die_Operatoren_des_Assemblers

<ausdruck1> * <ausdruck2> 167

Multiplikation <ausdruck1> mal <ausdruck2>

<ausdruck1> / <ausdruck2> 167

Division <ausdruck1> durch <ausdruck2>

<ausdruck1> + <ausdruck2> 167

Addition <ausdruck1> plus <ausdruck2>

<ausdruck1> - <ausdruck2> 167

Subtraktion <ausdruck1> minus <ausdruck2>

+ <ausdruck> 167

Das Vorzeichen von <ausdruck> wird beibehalten

- <ausdruck> 167

Das Vorzeichen von <ausdruck> wird umgekehrt

<segmentregister>:<ausdruck> 173

Das Standardsegmentregister fuer <ausdruck> wird durch <segmentregister> ueberschrieben

<segmentname>:<ausdruck> 157,173

Das Standardsegment fuer <ausdruck> wird durch <segmentname> ueberschrieben

<gruppenname>:< ausdruck> 173

Das Standardsegment fuer <ausdruck> wird durch <gruppenname> ueberschrieben

<variable>.<feld> 163,173

Addiert den Offset von <feld> zum Offset von <variable>

<ausdruck1>[<ausdruck2>] 160,170

Addiert den Wert von <ausdruck1> zu dem Wert von <ausdruck2>

&<symbolischer parameter> 207

Ersetzt <symbolischer parameter> durch den aktuellen Wert

<symbolischer parameter>& 207

Ersetzt <symbolischer parameter> durch den aktuellen Wert

<text> 209

Textliteral

!<zeichen> 209

Zeichenliteral

%<text> 209

Bewertet <text> nicht als Zeichenkette, sondern als Ausdruck und verwendet seinen Wert

;;<text> 210

Makrokommentar; wird nicht in die Makroerweiterung aufgenommen

<ausdruck1> AND <ausdruck2> 170

Bitweise logische UND-Verknuepfung

<zaehler> DUP [(<initwert>)[,(<initwert>,...]] 145

Spezifiziert mehrfaches Auftreten von einem oder mehreren Initialisierungswerten

<ausdruck1> EQ <ausdruck2> 169

Der Vergleich von <ausdruck1> und <ausdruck2> ergibt bei Gleichheit den Wert wahr (0FFFFh) zurueck

<ausdruck1> GE <ausdruck2> 169

Der Vergleich ergibt den Wert wahr (0FFFFh), wenn <ausdruck1> groesser oder gleich <ausdruck2> ist

<ausdruck1> GT <ausdruck2> 169

Der Vergleich ergibt den Wert wahr (0FFFFh), wenn <ausdruck1> groesser als <ausdruck2> ist

HIGH <ausdruck> 175

Gibt die hoeherwertigen 8 Bits von <ausdruck> zurueck

<ausdruck1> LE <ausdruck2> 169

Der Vergleich ergibt den Wert wahr (0FFFFh), wenn <ausdruck1> kleiner oder gleich <ausdruck2> ist

LENGTH <variable> 178

Liefert als Ergebnis die Anzahl BYTE-, WORD-, DWORD-, QWORD- oder TBYTE-Elemente in <variable>

LOW <ausdruck> 175

Gibt die niederwertigen 8 Bits von <ausdruck> zurueck

<ausdruck1> LT <ausdruck2> 169

Der Vergleich ergibt den Wert wahr (0FFFFh), wenn <ausdruck1> kleiner als <ausdruck2> ist

MASK <recordfeldname> 179

Gibt eine Bitmaske zurueck. Ein Bit ist 1, wenn das Bit zum Feld gehoert. Alle anderen Bits sind 0.

MASK <record> 179

Gibt eine Bitmaske zurueck. Ein Bit ist 1, wenn das Bit zum Record gehoert. Alle anderen Bits sind 0.

<ausdruck1> MOD <ausdruck2> 167

Rest nach der Division (Modulo)

<ausdruck1> NE <ausdruck2> 169

Der Vergleich von <ausdruck1> und <ausdruck2> gibt bei Ungleichheit den Wert wahr (0FFFFh) zurueck

NOT <ausdruck> 170

Bitweise logische Negation von <ausdruck>

OFFSET <ausdruck> 176

Gibt den Offset von <ausdruck> zurueck

<ausdruck1> OR <ausdruck2> 170

Bitweise logische ODER-Verknuepfung

<typ> PTR <ausdruck> 171

Weist <ausdruck> den durch <typ> spezifizierten Typ zu

SEG <ausdruck> 175

Gibt den Segmentwert von <ausdruck> zurueck

<ausdruck> SHL <anzahl> 168

<ausdruck> nach links um <anzahl> Bits verschieben

SHORT <marke> 174

Setz den Typ der <marke> auf SHORT; Verwendung in JMP-Befehlen, wenn die Entfernung zwischen -128 und + 127 liegt

<ausdruck> SHR <anzahl> 168

<ausdruck> nach rechts um <anzahl> Bits verschieben

SIZE <variable> 178

Gibt die totale Anzahl Bytes von <variable> zurueck

THIS <typ> 174

Erzeugt einen Operanden, dessen Offset und Segmentwert gleich dem aktuellen Speicherplatzzaehlerwert und dessen Typ der angegebene <typ> ist

TYPE <ausdruck> 176
gibt den Typ von <ausdruck> zurueck

.TYPE <ausdruck> 177

Gibt ein Byte zurueck, das den Modus und die Reichweite von <ausdruck> definiert

WIDTH <recordfeldname> 178

Gibt die Groesse eines Recordfeldes in Bits an

WIDTH <record> 179

Gibt die Groesse eines Records in Bits an

<ausdruck1> XOR <ausdruck2> 170

Exklusives ODER

Anhang D

CREF-Fehlermeldungen

Von CREF werden folgende Fehlermeldungen angezeigt:

can't open cross-refence file for reading

Die Datei mit der Erweiterung .CRF kann nicht gefunden werden. Es sind die korrekte Schreibweise des Namens und das spezifizierte Laufwerk zu pruefen.

can't open listing file for writing

Ursache fuer diese Fehlermeldung kann sein:

- Diskette ist voll oder schreibgeschuetzt
- eine Datei mit dem spezifizierten Namen ist schon vorhanden
- das angegebene Laufwerk ist nicht verfuegbar

cref has no switches

In der Befehlszeile von CREF wurde durch Schraegstrich (/) oder Bindestrich (-) ein Schalter spezifiziert. CREF besitzt keine Schalter.

extra file name ignore

In der Befehlszeile wurden mehr als zwei Dateinamen angegeben. CREF verwendet nur die ersten beiden Namen.

line invalid, start again

In der Befehlszeile bzw. nach der Eingabeaufforderung wurde keine Datei mit der Erweiterung .CRF eingegeben. Nach dieser Meldung erscheint die Aufforderung, diesen Dateinamen einzugeben.

out of heap space

CREF hat nicht genug Speicherplatz. Es sind residente Programme zu entfernen.

premature eof

Es wurde entweder keine gueltige Crossreferenzdatei spezifiziert, oder die Datei ist beschaedigt.

read error on stdio

Dieser Fehler tritt nur auf, wenn das Programm von der Tastatur oder aus einer Umlenkungsdatei ein <CTRL>+<Z> erhaelt.

1. 62. 548309. 3 D